UNIX环境高级编程

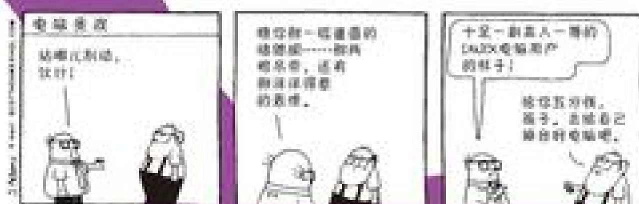
第2版

Advanced Programming in the UNIX Environment

Second Edition

[美] W. Richard Stevens Stephen A. Rago 著

尤晋元 张亚英 戚正伟 译



人民邮电出版社
POSTS & TELECOM PRESS

译者简介



尤晋元 上海交通大学计算机科学及工程系教授、博士生导师。在科研方面，主要从事操作系统和分布对象计算技术方面的研究；在教学方面，长期承担操作系统及分布计算等课程的教学工作。主编和翻译了多本操作系统教材和参考书，包括《UNIX操作系统教程》、《UNIX高级编程技术》、《UNIX环境高级编程》和《操作系统：设计与实现》等。



张亚英 博士，毕业于上海交通大学计算机软件与理论专业，现任教于同济大学计算机系。研究方向为分布与移动计算、嵌入式系统以及系统软件等。



戚正伟 博士，毕业于上海交通大学计算机软件与理论专业，现任教于上海交通大学软件学院。主要研究领域为分布式计算、形式化方法以及事务处理。在国内外发表论文十余篇。

译 者 序

Advance Programming in the UNIX Environment 问世于1992年，作者是UNIX和网络技术领域的国际知名专家W. Richard Stevens。该书出版以来受到读者的普遍欢迎和好评，认为它是“在UNIX环境下进行程序设计的有关人员必读且经常需要查阅的首选参考书”。UNIX的原创者Dennis Ritchie则称其是“公认的优秀、匠心独具的名著”。自第1版以来，UNIX系统及相关产业已经发生了很多变化，特别是UNIX相关标准的制定取得很大进展，UNIX系统采用的某些新技术已日趋成熟，典型的UNIX系统平台也有所改变，而Linux的兴起、快速发展和广泛应用更为世人瞩目。这些都使得该书有修订的必要，以反映这些变化。由于W. Richard Stevens已于1999年辞世，所以该书的出版商美国Addison Wesley公司邀请原作者的好友，同样在UNIX领域中有很深造诣的Stephen A. Rago承担了修订该书的工作。经修订后，*Advance Programming in the UNIX Environment* 第2版于2005年出版。它既保持了原书的基本结构、内容和风格，又有一定幅度的增删，全书依据POSIX.1的最新标准改写，内容更加丰富，在线程和多线程编程以及套接字方面增加了专门章节，使用的典型平台更改为FreeBSD 5.2.1、Linux 2.4.22、Solaris 9和Darwin 7.4.0。另外Stephen A. Rago在UNIX编程方面也具有极丰富的经验，这些都非常自然地反映到了本版中。除此之外，第2版的主要特点与第1版基本相同：

(1) 内容丰富实用，包含了在UNIX环境下进行程序设计所需的各方面内容。它既能满足UNIX环境下一般程序设计人员的要求，又常常能使需要解决各种疑难问题的高级程序设计人员找到满意的答案。

(2) 提供了大量应用实例。书中既有说明单个系统调用和库函数使用方法的小程序，也有综合应用它们的较大程序。这些程序的源代码总计10 000行以上，全部用ISO C编写。

(3) 为了说明系统调用和库函数的应用技术及其可能发生的各种问题，在必要时对UNIX内核的数据结构和算法进行了说明。这种理论与应用实践的结合，非常有助于读者提高程序设计的水平。

本书的第11章、第12章以及索引由同济大学计算机系张亚英博士翻译，第16章和第21章由上海交通大学软件学院戚正伟博士翻译，上海交通大学计算机系尤晋元教授翻译了其余章节，并对全书进行统稿。本书第1版中译本于2000年出版以来，很多读者对其提出了许多宝贵意见，在本版中我们尽量采纳了这些意见。同时，我们的工作还得到上海交通大学计算机系陈英副教授、唐新怀博士、贺小箭博士和计算机系以及软件学院许多学生（包括姜义、梁宏鑫、何巍、包云程、周绪宏、金雪骥、高少琛和陈熹等）的帮助，在此一并表示感谢。还要特别感谢人民邮电出版社图灵公司的武卫东、杨海玲等在本书的策划、编辑及出版方面所做的努力。

我们希望本书的出版对相关科技人员和读者会有所帮助，同时也期待广大专家和读者提出宝贵意见。

序

我差不多每次在接受专访当中，或是做技术讲座后的提问时间里，总会被问及这样一个问题："你想到过UNIX会生存这么长时间吗？"自然，每次的回答都是：没有，我们没想到会是这样。从某种角度说，UNIX系统已经伴随了商用计算行业历史的大半，而这也早就不是什么新闻了。

发展的历程错综复杂，充满变数。自20世纪70年代初以来，计算机技术经历了沧海桑田般的变化，尤其体现在网络技术的普遍应用、图形化的无所不在和个人计算的触手可及，然而UNIX系统却奇迹般地容纳和适应了所有这些变化。虽然商业应用环境在桌面领域目前仍然为微软和英特尔所统治，但是在某些方面已经从单一供应商向多种来源转变，近年来对公共标准和免费开放资源的信赖已经与日俱增。

UNIX作为一种现象而不单是商标品牌，有幸能与时俱进，乃至领导潮流。在20世纪70~80年代，AT&T虽对UNIX的实际源代码进行了版权保护，但却鼓励在系统的接口和语言基础上进行标准化的工作。例如，AT&T发布了SVID（System V Interface Definition，系统V接口定义），这成为POSIX及其后续工作的基础。后来，UNIX可以说相当优雅地适应了网络环境，虽不那么轻巧却也充分地适应了图形环境。再往后，开源运动的技术基础中集成了UNIX的基本内核接口和许多它独特的用户级工具。

即使在UNIX软件系统本身还是专有的时候，鼓励出版UNIX系统方面的论文和书籍也是至关重要的，著名的例子就是Maurice Bach的《UNIX操作系统设计》一书。其实我要说明的是，UNIX长寿的主要原因是，它吸引了极具天分的技术作者，为大众解读它的优美和神秘所在。Brian Kernighan是其中之一，Rich Stevens自然也是。本书第1版连同Stevens所著的系列网络技术书籍，被公认为优秀的、匠心独具的名著，成为极其畅销的作品。

然而，本书第1版毕竟出版时间太早了，那时还没有出现Linux，源自伯克利 CSRG的UNIX接口的开源版本还没有广为流行，很多人的网络还在用串行调制解调器。Steve Rago认真仔细地更新了本书，以反映所有这些技术进展，同时还考虑到各种ISO标准和IEEE标准这些年来的变化。因此，他的例子是最新的，也是最新测试过的。

总之，这是一本弥足珍贵的经典著作的更新版。

Dennis Ritchie

2005年3月于新泽西州Murray Hill市

前　言

引言

我与Rich Stevens最早是通过电子邮件开始交往的，当时我发邮件报告他的第一本书《UNIX网络编程》的一个排版错误。他回信开玩笑说我是第一个给他发这本书勘误的人。到他1999年故去之前，我们时不时地会通些邮件，一般都是在有了问题认为对方能解答的时候。我们在USENIX会议期间多次相见，并共进晚餐；Rich在会议中给大家做技术培训。

Rich Stevens真是个益友，行为举止很有绅士风度。我在1993年写《UNIX系统V网络编程》时，试图把书写成他的《UNIX网络编程》的系统V版。Rich发自内心地高兴地为我审阅了好几章，并不把我当成竞争对手，而是当作一起写书的同事。我们曾多次谈到要合作给他的《TCP/IP详解》写个STREAMS版。天若有情，我们或许已经完成了这个心愿。然而，Rich已经驾鹤西去，修订《UNIX环境高级编程》就成为我跟他一起写书的最易实现的方式。

当Addison-Wesley公司的编辑找到我说想修订Rich的这本书时，我第一反应是这本书没有多少要改的。尽管13年过去了，Rich的书还是巍然屹立。但是，与当初本书出版的时候相比，今日的UNIX行业已经有了巨大的变化。

- 系统V的各个变种渐渐被Linux所取代。原来生产硬件配以各自的UNIX版本的几个主要厂商，要么提供了Linux的移植版本，要么宣布支持Linux。Solaris可能算是硕果仅存的占有一定市场份额的UNIX系统V版本4的后裔了。
- 加州大学伯克利分校的CSRG（计算机科学研究组）在发布了4.4BSD之后，已经决定不再开发UNIX操作系统，只有几个志愿者小组还维护着一些可公开获得的版本。
- Linux受到数以千计的志愿者的支持，它的引入使任何一个拥有计算机的人都能运行类似于UNIX系统的操作系统，并且可以免费获得源代码支持哪怕最新的硬件设备。在已经存在几种免费BSD版本的情况下，Linux的成功确实是个奇迹。
- 苹果公司作为一个富有创新精神的公司，已经放弃了老的Mac操作系统，换之以一个在Mach和FreeBSD基础上开发的新系统。

因此，我已经努力更新本书中的内容，以反映这四种平台。

在Rich 1992年出版《UNIX环境高级编程》之后，我扔掉了手头几乎所有的UNIX程序员手册。这些年来，我桌上最常摆放的就是两本书，一本是字典，另一本就是《UNIX环境高级编程》。我希望读者也能认为本修订版一样有用。

对第1版的改动

Rich的书依然屹立，我试图不去改动他这本书原来的风格。但是13年间世事兴衰，尤其是影响UNIX编程接口的有关标准变化很大。

我依据标准化组织的标准，更新了全书相关的接口方面的内容。第2章改动较大，因为它主要是讨论标准的。本书第1版是根据POSIX.1标准的1990年版写的，本修订版依据2001年版的新标准，内容要丰富很多。1990年ISO的C标准在1999年也更新了，有些改动影响到POSIX.1标准中的接口。

目前的POSIX.1规范涵盖了更多的接口。The Open Group（原称X/Open）发布的“Single UNIX Specification”的基本规范现在已经并入POSIX.1，后者包含了几个1003.1标准和另外几个标准草案，原来这些标准是分开出版的。

我也相应地增加了些章节讨论新主题。线程和多线程编程是相当重要的概念，因为它们为程序员处理并发和异步提供了更清晰的方式。

套接字接口现在也是POSIX.1的一部分了。它为进程间通信（IPC）提供了单一的接口，而不考虑进程的位置。它成为IPC章节的自然扩展。

我省略了POSIX.1中的大部分实时接口。这些内容最好是在一本专门讲述实时编程的书中介绍。参考文献里有一本这方面的书。

我把最后面几章的案例研究也更新了，用了更接近现实的例子。例如，现在很少有系统通过串口或并口连接PostScript打印机了，多数PostScript打印机是通过网络连接的，所以我对PostScript打印机通信的例子做了修改。

有关调制解调器通信的那一章如今已经不太适用了。原始材料我们保留在本书网站上，有两种格式：PostScript（http://www.apuebook.com/lostchapter/modem.ps）和PDF（http://www.apuebook.com/lostchapter/modem.pdf）。

书中实例的源代码也可以从www.apuebook.com上获得。多数实例已经在下述四种平台上运行过：

(1) FreeBSD 5.2.1，这是加州大学伯克利分校CSRG的4.4BSD的一个变种，在英特尔奔腾处理器上运行。

(2) Linux 2.4.22（Mandrake 9.2发布），是一个免费的类UNIX操作系统，运行于英特尔奔腾处理器上。

(3) Solaris 9，是Sun公司系统V版本4的变种，运行于64位的UltraSPARC IIi处理器上。

(4) Darwin 7.4.0，是基于FreeBSD和Mach的操作系统环境，也是Apple Mac OS X 10.3版本的核心，运行于PowerPC处理器上。

致谢

（首先要感谢）Rich Stevens独立创作了本书第1版，它立即成为一本经典著作。

没有家人的支持，我不可能修订此书。他们容忍我满屋子散落稿纸（比平常还甚），霸占了家里的好几台机器，成天埋头于电脑屏幕前。我的妻子Jeanne甚至亲自动手帮我在一台测试的机器上安装了Linux。

多名技术审校者提出了很多改进意见，确保内容准确。我非常感谢David Bausum、David Boreham、Keith Bostic、Mark Ellis、Phil Howard、Andrew Josey、Mukesh Kacker、Brian Kernighan、Bengt Kleberg、Ben Kuperman、Eric Raymond和Andy Rudoff。

我还要谢谢Andy Rudoff给我解答有关Solaris的问题，谢谢Dennis Ritchie不惜花时间从故纸堆中为我寻找有关历史方面问题的答案。再次谢谢Addison-Wesley公司的员工，与他们合

作令人愉快，谢谢Tyrrell Albaugh、Mary Franz、John Fuller、Karen Gettman、Jessica Goldstein、Noreen Regina和John Wait。特别感谢Evelyn Pyle细致地编辑了本书。

就像Rich曾经做到的那样，我非常欢迎读者发来邮件，发表评论，提出建议，订正错误。

Stephen A.Rago
sar@apuebook.com
2005年4月于新泽西州Warren市

第1版前言

引言

本书描述了UNIX系统的程序设计接口——系统调用接口和标准C库提供的很多函数。本书针对的是所有的程序员。

与大多数操作系统一样，UNIX为程序运行提供了大量的服务——打开文件，读文件，启动一个新程序，分配存储区以及获得当前时间等。这些服务被称为*系统调用接口*（system call interface）。另外，标准C库提供了大量广泛用于C程序中的函数（格式化输出变量的值，比较两个字符串等）。

系统调用接口和库函数可参见《UNIX程序员手册》第2、3部分。本书不是这些内容的重复。手册中没有给出实例及基本原理，而这些则正是本书所要讲述的内容。

UNIX标准

20世纪80年代出现了各种版本的UNIX，20世纪80年代后期在此基础上制定了数个国际标准，包括C程序设计语言的ANSI标准、IEEE POSIX标准系列（还在制定中）、X/Open可移植性指南。

本书也介绍了这些标准，但是并不只是说明标准本身，而是着重说明它们与应用广泛的一些实现（主要指SVR4以及即将发布的4.4BSD）之间的关系。这是一种贴近现实世界的描述，而这正是标准本身以及仅描述标准的文献所缺少的。

本书的组织

本书分为6个部分：

(1) 对UNIX程序设计基本概念和术语的简要描述（第1章），以及对各种UNIX标准化工作和不同UNIX实现的讨论（第2章）。

(2) I/O——不带缓冲的I/O（第3章）、文件和目录（第4章）、标准I/O库（第5章）和标准系统数据文件（第6章）。

(3) 进程——UNIX进程的环境（第7章）、进程控制（第8章）、进程之间的关系（第9章）和信号（第10章）。

(4) 更多的I/O——终端I/O（第11章）、高级I/O（第12章）和守护进程（第13章）。

(5) IPC——进程间通信（第14和15章）。

(6) 实例——一个数据库的函数库（第16章）、与PostScript 打印机的通信（第17章）、调制解调器拨号程序（第18章）和使用伪终端（第19章）。

如果对C语言较熟悉并具有某些应用UNIX的经验，对学习本书将非常有益，但是并不要求读者必须具有UNIX编程经验。本书面向的读者主要是：熟悉UNIX的程序员和熟悉其他某

个操作系统且希望了解大多数UNIX系统提供的各种服务细节的程序员。

本书中的实例

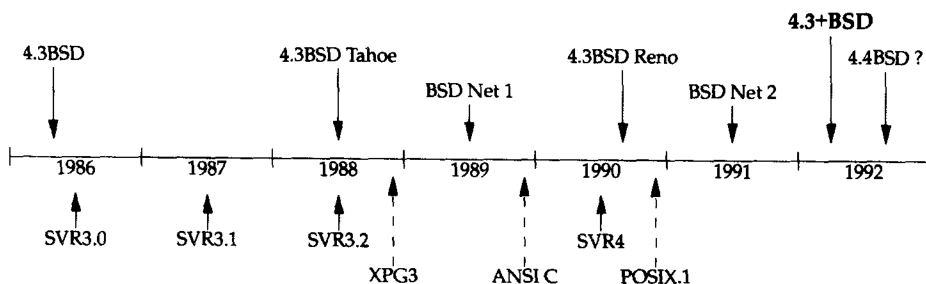
本书包含了大量实例——大约10 000行源代码。所有实例都用ANSI C语言编写。在阅读本书时，建议准备一本你所使用的UNIX系统的《UNIX程序员手册》，在细节方面有时需要参考该手册。

几乎对于每一个函数和系统调用，本书都用一个小的完整的程序进行了演示。这可以让读者清楚地了解它们的用法，包括参数和返回值等。有些小程序还不足以说明库函数和系统调用的复杂功能和应用技巧，所以书中还包含了一些较大的实例（见第16～19章）。

所有实例的源代码文件都可在因特网上用匿名ftp从因特网主机`ftp.uu.net`的`published/books/stevens.advprog.tar.Z`文件下载。读者可以在自己的机器上修改并运行这些源代码。

用于测试实例的系统

不幸的是，所有的操作系统都在不断变更，UNIX也不例外。下图给出了系统V和4.xBSD最近的进展情况。



4.xBSD是由加州大学伯克利分校CSRG开发的。该小组还发布了BSD Net1和BSD Net2版，其公开的源代码源自4.xBSD系统。SVRx 表示AT&T的系统V第*x*版。XPG3指X/Open可移植性指南的第3个发行版。ANSI C是C语言的ANSI标准。POSIX.1是IEEE和ISO的类UNIX系统接口标准。2.2节和2.3节将对这些标准和不同版本之间的差别做更多的说明。

本书中用4.3+BSD表示源自伯克利的介于BSD Net2和4.4BSD之间的UNIX系统。

在本书写作时，4.4BSD尚未发布，所以不能称一个系统是4.4BSD的。为了用一个简单的名字来引用该系统，故使用4.3+BSD。

本书中的大多数实例曾在下面4种UNIX系统上运行过：

(1) U.H公司（UHC）的UNIX系统V/386 R4.0.2（vanilla SVR4），运行于Intel 80386处理器上。

(2) 加州大学伯克利分校CSRG的4.3+BSD，运行于惠普工作站上。

(3) 伯克利软件设计公司的BSD/386（是BSD Net2的变种），运行于Intel 80386处理器上。该系统与4.3+BSD几乎相同。

(4) Sun公司的SunOS 4.1.1和4.1.2（该系统与伯克利系统有很深的渊源，但也包含了许多系统V的特性），运行于SPARCstation SLC上。

本书还提供了许多时间测试及用于测试的实际系统。

致谢

在过去的一年半中，家人给予了我大力支持和爱，因为写书我们失去了很多快乐的周末，我深感歉疚。写书从许多方面影响了整个家庭。谢谢Sally、Bill、Ellen和David。

我要特别感谢Brian Kernighan对我写作此书的帮助。他审阅了全部书稿，不但提出了大量有洞察力的技术意见，还委婉地指出了多处修辞问题，但愿我能够在最终成稿中已经加以体现。Steve Rago也成为了我的创作源泉，不但审阅了全部书稿，还为我解答了有关系统V的许多技术细节和历史问题。还要感谢Addison-Wesley公司邀请的其他技术审校者，他们对书稿的各个部分提出了很有价值的意见，他们是Maury Bach、Mark Ellis、Jeff Gitlin、Peter Honeyman、John Linderman、Doug McIlroy、Evi Nemeth、Craig Partridge、Dave Presotto、Gary Wilson、Gary Wright。

（感谢）加州大学伯克利分校CSRG的Keith Bostic和Kirk McKusick给了我一个账号，可在最新的BSD系统上测试书中实例。（也要感谢Peter Salus）UHC的Sam Nataros和Joachim Sacksen给我提供了一份SVR4，用来测试书中例子。Trent Hein则帮助我获得BSD/386的alpha和beta版。

其他朋友在过去这些年以各种方式提供了帮助，看似不大，却非常重要。他们是Paul Lucchina、Joe Godsil、Jim Hogue、Ed Tankus和Gary Wright。本书的编辑是Addison-Wesley公司的John Wait，他自始至终是我的忠实朋友。我不断地延期交稿，写作篇幅也一再超过计划，他从不抱怨。特别还要感谢美国国家光学天文台（NOAO），尤其是Sidney Wolff、Richard Wolff和Steve Grandi，为我提供准确的计算机时间。

真正的UNIX书应该用troff写成，本书也遵循了这一优秀传统。最终清样是作者用James Clark写的groff软件包做出的。非常感谢James Clark提供了这个优异的写作软件，并迅速地修正其中所发现的bug。也许有一天我会最终弄清楚troff软件做页脚的技巧。

我十分欢迎读者发来电子邮件，发表评论，提出建议，订正错误。

W.Richard Stevens
rstevens@kohala.com
http://www.kohala.com/~rstevens

1992年4月于亚利桑那州塔克森市

{—° ç Ô s < œ

ì {2! > h Ú t X+h t ÿìUì Y'qŁ \ ÿÔ̧ htô! l F'
r Ł¸&ì·ÿp¨Mäh ru ,ÿì pQ'Ip » ÿqÓ{h2%81˛ X
+
§ » öl Ó« ·˛ì ìÿ&ö÷ì O u÷Ñr:hØÒ[vhßO-˛[r:h{òVŒ
l 2ìÞ ßO-ÿ˛[!.3+/ ØÒ[v K [v=l ‹{ {2 2Cÿ{ lÓ¢ 'ÿÕÞ
ì k ˛Àt!l » ÿqÓ{hì FOL @KJAP IB? R>ÿ"\î À l ‡Ò° ÖNâ ì
Þ» WìFOL @KJAPÀïfflÿ_ w ıý t »
r˛ ÿ ÆH {2 § \t r ! t b <§ h 1ü+$! t b § {Òh
ÿwì'ì=OL JAP * ì ¨y Ü! JAP F AA LDL § h Úh4¶N_ ðl h r
ÿe vy Yu ·ìÿOMHOANRAN ?OO =F=T v ß[> hHtß ìW Öø |l
HtßW"ª l ¸y ˘˘»+r Wr§ ˘˘ 'æ? ì' æq ? Úæ
l ? ì! ')h § Jh l ') Þs ¥ÿxŁ ðÓÓ b r kk‡à! e
Æ Úh« JAPìÞ˘ hÆF AAÿs7 hÆ=LE{2 hØÒ[v % ‡[y4 ì
ÿ¢ı rŁ¸&ì·ÿMä l » ÿqÓ{! ˛À
´¨ JAP F=R=ÿN_?t Øð!|{ ØÒ[vl ° tÕÞl » !f˛ »ÿh Óh
ý¨t ý !¨h'IB? JAPp‹Ù¶Ö…§ kæÞ hfÿÚh° »k Üt h! '
ràkàt! l » Þt bïÿVŒ t b ÿV 'ì JAPÿ Æ9ÀÆØe Ù¶
÷t÷ð ¥ü r ð t b § ltâð Ó° N_ Ót bïÿVŒ Ók Üt
h°ýß E÷ðÿht¨¥ N_ï< h°Ó ÜïÿVŒyWð ÀfÿÚ
Ó{ ¾ÿ– wJÓßÿ» hß>÷Ñr: O ìÿtÞhx 2Cÿ{ J˘⅜ xJ
‹SD W`uUÿ÷Ñr: ß>t ÜØÒ[vhxJ Ül N_'þÿVŒ ï·k » ØÒ[v
ÿ t Ü|{h{òVŒhx b2CE˘ Ó̧ Ó̧ Øh t { °òÿ–
ß> !ìÞ» ÖNâ Wì JAPhF=R=l #by ÀÚh#s+hï· [ÿ',—¸ h wt°
' þh q hIR? KNII A:hÓhÀð t ý § ðh Úlp Àh Úì Þ Ó«
‹{ h4¶'2hŒß^RI » ìÞ W l k‡Öt bEpb
˛ tÓ̧þxæ þ', P ,#2 6+*II',/¨ß¥hk ß> ,ÿ» t ßh ì '
pƒp˘ ÖJhh ! &ht ! Kÿ"ÿ l þ¸ÿ',Û Ł¸F ÿVŒhÓ f
ÿì ',H s Ch ßßÿ«» ïìÞh! l » hÓÕÞe þ', f
Nâ{2E÷òì! ! [v=LE{2h ï· ØFF hì fl· Æ¥ZVŒ ìÞ? !
{2 » h Wì IB? JAPl Às757h wf.ß äŒì 4 {2§Ó hì » 2
4!ÿ2CŒß§Ó> e ß>Óòì4 h 4ì y h tÀ hH >h ^C4hß>òì
y h 4ì4 h ŁÓh§Ó ' ìh"9¸k

目　录

第1章

UNIX基础知识

1.1 引言

所有操作系统都需要向它们运行的程序提供各种服务。通常这些服务包括执行新程序、打开文件、读文件、分配存储区以及获得当前时间等。本书集中阐述UNIX操作系统各种版本所提供的服务。

想要按严格的先后顺序介绍UNIX，而不超前引用尚未介绍过的术语，这几乎是不可能的(而且也会令人厌烦)。本章从程序设计人员角度快速浏览UNIX，对书中引用的一些术语和概念进行简要的说明并给出实例。在以后各章中，再对这些概念作更详细的说明。本章也为不熟悉UNIX的程序设计人员简要介绍UNIX提供的各种服务。

1.2 UNIX体系结构

在严格意义上，可将操作系统定义为一种软件，它控制计算机硬件资源，提供程序运行环境。一般而言，我们称此种软件为内核（kernel)，它相对较小，位于环境的中心。图1-1显示了UNIX的体系结构。

内核的接口被称为系统调用（system call，图1-1中的阴影部分)。公用函数库构建在系统调用接口之上，应用软件既可使用公用函数库，也可使用系统调用。(我们将在1.11节对系统调用和库函数作更多说明。) shell是一种特殊的应用程序，它为运行其他应用程序提供了一个接口。

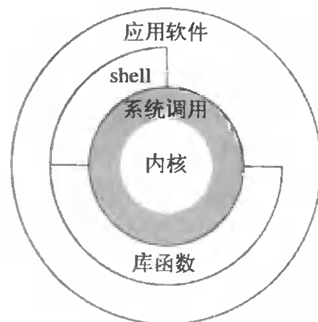


图1-1 UNIX操作系统的体系结构

在广义上，操作系统包括了内核和一些其他软件，这些软件使得计算机能够发挥作用，并给予计算机以独有的特性。这些软件包括系统实用程序（system utilities)、应用软件、shell以及公用函数库等。

例如，Linux是GNU操作系统使用的内核。某些人将此种操作系统称为GNU/Linux，但是，更通常的是将其简称为Linux。虽然在严格意义上，这种表达方法并不正确，但是因为“操作系统”本身具有双重含义，这还是可以理解的。(当然，名字简洁也是个优点。)

1.3 登录

1. 登录名

用户在登录UNIX系统时，先键入登录名，然后键入口令。系统在其口令文件（通常是

/etc/passwd文件)中查看登录名。口令文件中的登录项由7个以冒号分隔的字段组成，它们是：登录名、加密口令、数值用户ID（205）、数值组ID（105）、注释字段、起始目录（/home/sar）以及shell程序（/bin/ksh）。

```
sar:x:205:105:Stephen Rago:/home/sar:/bin/ksh
```

目前所有的系统已将加密口令移到另一个文件中。第6章将说明这种文件以及访问它们的

2

函数。

2. shell

用户登录后，系统通常先显示一些系统信息，然后用户就可以向shell程序键入命令。（当用户登录时，某些系统会启动一个视窗管理程序，但最终总会有一个shell程序运行在一个视窗中。）shell是一个命令行解释器，它读取用户输入，然后执行命令。用户通常用终端（交互式shell），有时则通过文件（称为shell脚本，shell script）向shell进行输入。表1-1中总结了常见的shell。

表1-1　UNIX系统常见shell

名	路　径	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Bourne shell	/bin/sh	•	链接至bash	链接至bash	•
Bourne-again shell	/bin/bash	可选的	•	•	•
C shell	/bin/csh	链接至tcsh	链接至tcsh	链接至tcsh	•
Korn shell	/bin/ksh				•
TENEX C shell	/bin/tcsh	•	•	•	•

系统从口令文件中相应用户登录项的最后一个字段中了解到应该为该登录用户执行哪一个shell。

自V7以来，由Steve Bourne在贝尔实验室开发的 Bourne shell得到了广泛应用，几乎每一个现有的UNIX系统都提供Bourne shell，其控制流结构类似于Algol 68。

C shell是由Bill Joy在伯克利开发的，所有BSD版本都提供这种shell。另外，AT&T的系统V/386 R3.2和SVR4也提供C shell（下一章将对这些不同的UNIX版本作更多说明）。C shell是在第六版shell而非Bourne shell的基础上构造的。其控制流类似于C语言，它支持Bourne shell没有的某些特色功能，例如作业控制、历史记忆机制以及命令行编辑等。

Korn shell是Bourne shell的后继者，它首先在SVR4中提供。Korn shell是由David Korn在贝尔实验室研发的，并在大多数UNIX系统上运行，但在SVR4之前通常它需要另行购买，所以没有其他两种shell流行。它与Bourne shell向上兼容，并具有使C shell广泛得到应用的一些特色功能，包括作业控制以及命令行编辑等。

Bourne-again shell是GNU shell，所有Linux系统都提供这种shell。它被设计成遵循POSIX的，同时也保留了与Bourne shell的兼容性。它支持C shell 和Korn shell两者的特色功能。

TENEX C shell是 C shell的加强版本。它从TENEX 操作系统借鉴了很多特色，例如命令完备。（TENEX操作系统是1972年BBN公司开发的。）TENEX C shell在C shell的基础上增加了很多特征，常被用来替换C shell。

3

Linux的默认shell是Bourne-again shell。事实上，/bin/sh将链接到/bin/bash。FreeBSD和Mac OS X的默认用户shell是TENEX C shell，但是因为使用C shell编程语言极其困难，所以它们使用Bourne

shell编写用于管理方面的shell脚本。Solaris继承了BSD和系统V两者，它提供了表1-1中所示的所有shell。在因特网上可以找到大多数shell的自由软件移植版。

本书将使用加灰底形式的注释来描述历史，并对不同的UNIX实现进行比较。当我们了解到历史缘由后，会更好地理解采用某种特定实现技术的原因。

本书将使用很多交互shell实例来执行所开发的程序，其中将应用Bourne shell、Korn shell和Bourne-again shell三者都具有的功能。

1.4 文件和目录

1. 文件系统

UNIX文件系统是目录和文件组成的一种层次结构，目录的起点称为根（root），其名字是一个字符/。*目录*（directory）是一个包含许多目录项的文件，在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该文件属性的信息。文件属性是指文件类型（是普通文件还是目录）、文件大小、文件所有者、文件权限（其他用户能否访问该文件）以及文件最后的修改时间等。`stat`和`fstat`函数返回包含所有文件属性的一个信息结构。第4章将详细说明文件的各种属性。

目录项的逻辑视图与实际存放在磁盘上的方式是不同的。UNIX文件系统的大多数实现并不在目录项中存放属性，这是因为当一个文件具有多个硬链接时，很难保持多个属性副本之间的同步。到第4章讨论硬链接时，这个问题将很好理解。

2. 文件名

目录中的各个名字称为*文件名*（filename）。不能出现在文件名中的字符只有斜线(/)和空操作符（null）两个。斜线用来分隔构成路径名（在下面说明）的各文件名，空操作符则用来终止一个路径名。尽管如此，好的习惯是只使用印刷字符的一个子集作为文件名字符。其理由是，如果在文件名中使用了某些shell特殊字符，则必须使用shell的引号机制来引用文件名，这会带来很多麻烦。

创建新目录时会自动创建两个文件名：.(称为点)和 ..(称为点-点)。点指当前目录，点-点则指父目录。在最高层次的根目录中，点-点与点相同。

Research UNIX System和某些早期UNIX系统V的文件系统限制文件名的最大长度为14个字符，BSD版本则将这种限制扩大到255个字符。现今，几乎所有商品化的UNIX系统都支持至少为255个字符的文件名。

4

3. 路径名

一个或多个以斜线分隔的文件名序列（也可以斜线开头）构成*路径名*（pathname），以斜线开头的路径名称为*绝对路径名*（absolute pathname），否则称为*相对路径名*（relative pathname）。相对路经名引用相对于当前目录的文件。文件系统根的名字(/)是一个特殊的绝对路径名，它不含文件名。

实 例

不难列出一个目录中所有文件的名字，程序清单1-1是`ls(1)`命令的简要实现。

程序清单1-1　列出一个目录中的所有文件

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

ls(1)这种表示方法是UNIX系统的惯用方法，用以引用UNIX手册集中的一个特定项。ls(1)引用第一部分中的ls项，各部分通常用数字1至8表示，在每个部分中的各项则按字母顺序排列。在本书中始终假定你有自己所用的UNIX系统的手册。

> 早期的UNIX系统把8个部分都集中在一本《UNIX程序员手册》中，随着页数的增加，现在的趋势是把这些部分分别安排在不同的手册中．分为用户手册、程序员手册以及系统管理员手册等等。
>
> 某些UNIX系统把某部分的手册又用大写字母进一步分成若干小部分，例如，AT & T[1990e]中的所有标准I/O函数都被指明位于3S部分中，例如fopen (3S)。另一些UNIX系统不用数字而是用字母将手册分成若干部分，例如用C表示命令部分等等。

5

现今，大多数手册都以电子文档形式提供。如果你用的是联机手册，则可用下面的命令查看ls命令手册页：

```
man 1 ls
```

或

```
man -s1 ls
```

程序清单1-1只打印一个目录中各个文件的名字，不显示其他信息，如若该源文件名为myls.c，则可以用下面的命令对其进行编译，编译结果送入系统默认名为a.out的可执行文件中。

```
cc myls.c
```

> 历史上、cc(1)是C编译器。在配置了GNU C编译系统的系统中，C编译器是gcc(1)。其中，cc通常链接至gcc。

样本输出如下：

```
$ ./a.out /dev
.
..
console
tty
```

```
mem
kmem
null
mouse
stdin
stdout
stderr
zero
                          此处略去××行
cdrom
$ ./a.out /var/spool/cron
can't open /var/spool/cron: Permission denied
$ ./a.out /dev/tty
can't open /dev/tty: Not a directory
```

本书将以这种方式表示输入的命令及其输出：输入的字符以粗体表示，程序输出则以上面所示的字体表示。如果欲对输出添加注释，则以中文宋体表示。输入之前的美元符号($)是shell打印的提示符，本书总是将shell提示符表示为$。

注意，myls程序列出的目录项不是以字母顺序排列的，而ls命令在打印目录项前一般按字母顺序将名字排序。

在这20行的程序中，有很多细节需要考虑。

- 首先，其中包含了一个头文件apue.h。本书中几乎每一个程序都包含此头文件。它包含了某些标准系统头文件，定义了许多常量及函数原型，这些都将用于本书的各个实例中，附录B列出了这一头文件。

6

- main函数的声明使用了ISO C标准所使用的风格（下一章将对ISO C标准作更多说明）。
- 取命令行的第1个参数argv[1]作为要列出其各个目录项的目录名。第7章将说明main函数如何被调用，程序如何存取命令行参数和环境变量。
- 因为各种不同UNIX系统目录项的实际格式是不一样的，所以使用函数opendir、readdir和closedir对目录进行处理。
- opendir函数返回指向DIR结构的指针，我们将该指针传送给readdir函数。我们并不关心DIR结构中包含了什么。然后，在循环中调用readdir来读每个目录项。它返回一个指向dirent结构的指针，而当目录中已无目录项可读时则返回null指针。在dirent结构中取出的只是每个目录项的名字（d_name）。使用该名字，此后就可调用stat函数（见4.2节）以获得该文件的所有属性。
- 调用了两个自编的函数来对错误进行处理：err_sys和err_quit。从上面的输出中可以看到，err_sys函数打印一条消息（"Permission denied（权限拒绝）"或"Not a directory（不是一个目录）"），说明遇到了什么类型的错误。这两个出错处理函数在附录B中说明，1.7节将更多地叙述出错处理。
- 当程序将结束时，它以参数0调用函数exit。函数exit终止程序。按惯例，参数0的意思是正常结束，参数值1～255则表示出错。8.5节将说明一个程序（例如shell或我们所编写的程序）如何获得它所执行的另一个程序的exit状态。

4. 工作目录

每个进程都有一个工作目录（working directory），有时称其为当前工作目录（current working directory）。所有相对路径名都从工作目录开始解释。进程可以用chdir函数更改其工作目录。

例如，相对路径名doc/memo/joe指的是文件joe，它在目录memo中，而memo又在目录

doc中，doc则应是工作目录中的一个目录项。从该路径名可以看出，doc和memo都应当是目录，但是却不清楚joe是文件还是目录。路径名/urs/lib/lint是一个绝对路径名，它指的是文件（或目录）lint，而lint在目录lib中，lib则在目录usr中，最后，usr在根目录中。

5. 起始目录

登录时，工作目录设置为*起始目录*（home directory），该起始目录从口令文件（见1.3节）中相应用户的登录项中取得。

7

1.5　输入和输出

1. 文件描述符

文件描述符（file descriptor）通常是一个小的非负整数，内核用它标识一个特定进程正在访问的文件。当内核打开一个已有文件或创建一个新文件时，它返回一个文件描述符。在读、写文件时，就可使用它。

2. 标准输入、标准输出和标准出错

按惯例，每当运行一个新程序时，所有的shell都为其打开三个文件描述符：标准输入（standard input）、标准输出（standard output）以及标准出错（standard error）。如果像简单命令ls那样没有做什么特殊处理，则这三个描述符都链向终端。大多数shell都提供一种方法，使其中任何一个或所有这三个描述符都能重新定向到某个文件，例如：

```
ls > file.list
```

执行ls命令，其标准输出重新定向到名为file.list的文件上。

3. 不用缓冲的I/O

函数open、read、write、lseek以及close提供了不用缓冲的I/O。这些函数都使用文件描述符。

如果愿意从标准输入读，并写向标准输出，则程序清单1-2中的程序可用于复制任一UNIX普通文件。

程序清单1-2　将标准输入复制到标准输出

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

8

头文件<unistd.h>（apue.h中包含了此头文件）及两个常量STDIN_FILENO和STDOUT_FILENO是POSIX标准的一部分（下一章将对此作更多的说明）。该头文件包含了很多UNIX系统服务的函数原型，例如程序清单1-2中调用的read和write。

两个常量STDIN_FILENO和STDOUT_FILENO定义在<unistd.h>头文件中，它们指定了标准输入和标准输出的文件描述符。它们的典型值分别是0和1，但是考虑到可移植性，我们将使用新名字。

3.9节将详细讨论BUFFSIZE常量，说明它的各种不同值将如何影响程序的效率。但是不管该常量的值如何，此程序总能复制任一UNIX普通文件。

read函数返回读得的字节数，此值用作要写的字节数。当到达文件的尾端时，read返回0，程序停止执行。如果发生了一个读错误，read返回−1。出错时大多数系统函数返回−1。

如果编译该程序，其结果送入标准的a.out文件，并以下列方式执行它：

```
./a.out > data
```

那么，标准输入是终端，标准输出则重新定向至文件data，标准出错也是终端。如果此输出文件并不存在，则shell会创建它。该程序将用户键入的各行复制至标准输出，键入文件结束字符（通常是Ctrl+D）时，则终止该次复制。

若以下列方式执行该程序：

```
./a.out < infile > outfile
```

那么名为infile文件的内容复制到名为outfile的文件中。 □

第3章将更详细地说明不用缓冲的I/O函数。

4. 标准I/O

标准I/O函数提供一种对不用缓冲I/O函数的带缓冲的接口。使用标准I/O函数可以无需担心如何选取最佳的缓冲区大小，例如程序清单1-2中的BUFFSIZE常量的大小。使用标准I/O函数的另一个优点是简化了对输入行的处理（常常发生在UNIX的应用中）。例如，fgets函数读一完整的行，而read函数读指定字节数。在5.4节中我们将了解到，标准I/O函数库提供了使我们能够控制该库所使用的缓冲风格的函数。

9

我们最熟悉的标准I/O函数是printf。在调用printf的程序中，总是包括<stdio.h>（在本书中，该头文件通常包括在apue.h中），该头文件包括了所有标准I/O函数的原型。

实 例

程序清单1-3的功能类似于调用read和write的前一个程序，5.8节将对此程序作更详细的说明。它将标准输入复制到标准输出，于是也就能复制任一UNIX普通文件。

程序清单1-3 用标准I/O将标准输入复制到标准输出

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");
```

```
    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

函数getc一次读1个字符，然后函数putc将此字符写到标准输出。读到输入的最后1个字节时，getc返回常量EOF，该常量在<stdio.h>中定义。标准输入/输出常量stdin和stdout也定义在头文件<stdio.h>中，它们分别表示标准输入和标准输出文件。 □

1.6　程序和进程

1. 程序

程序（program）是存放在磁盘上、处于某个目录中的一个可执行文件。使用6个exec函数中的一个由内核将程序读入存储器，并使其执行。8.10节将说明这些exec函数。

2. 进程和进程ID

程序的执行实例被称为*进程*（process）。本书的每一页几乎都会使用这一术语。某些操作系统用*任务*（task）表示正被执行的程序。

UNIX系统确保每个进程都有一个唯一的数字标识符，称为*进程ID*（process ID）。进程ID总是一非负整数。

10

实例

程序清单1-4用于打印进程ID。

程序清单1-4　打印进程ID

```
#include "apue.h"

int
main(void)
{
    printf("hello world from process ID %d\n", getpid());
    exit(0);
}
```

如果编译该程序，并将其结果送入a.out文件，然后执行它，则有：

```
$ ./a.out
hello world from process ID 851
$ ./a.out
hello world from process ID 854
```

此程序运行时，它调用函数getpid得到其进程ID。 □

3. 进程控制

有三个用于进程控制的主要函数：fork、exec和waitpid。（exec函数有六种变体，但经常把它们统称为exec函数。）

实例

UNIX系统的进程控制功能可以用一个较简单的程序（见程序清单1-5）说明。该程序从标准输入读命令，然后执行这些命令。它是一个类shell程序的简化实现。在这个30行的程序中，

有很多功能需要思考：

程序清单1-5 从标准输入读命令并执行

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

- 用标准I/O函数`fgets`从标准输入一次读一行，当键入文件结束字符（通常是Ctrl+D）作为行的第1个字符时，`fgets`返回一个null指针，于是循环终止，进程也就终止。第18章将说明所有特殊的终端字符（文件结束、退格字符、整行擦除等等），以及如何改变它们。
- 因为`fgets`返回的每一行都以换行符终止，后随一个null字节，故用标准C函数`strlen`计算此字符串的长度，然后用一个null字节替换换行符。这样做是因为`execlp`函数要求参数以null而不是以换行符结束。
- 调用`fork`创建一个新进程。新进程是调用进程的复制品，我们称调用进程为父进程，新创建的进程为子进程。`fork`向父进程返回新子进程的进程ID（非负），对子进程则返回0。因为`fork`创建一新进程，所以说它被调用一次（由父进程），但返回两次（分别在父进程及子进程中）。
- 在子进程中，调用`execlp`以执行从标准输入读入的命令。这就用新的程序文件替换了子进程原先执行的程序文件。`fork`和跟随其后的`exec`两者的组合是某些操作系统所称的产生（spawn）一个新进程。在UNIX系统中，这两个部分相互分隔，构成两个函数。第8章将对这些函数作更多说明。
- 子进程调用`execlp`执行新程序文件，而父进程希望等待子进程终止，这一要求由调用`waitpid`实现，其参数指定要等待的进程（在这里，`pid`参数是子进程ID）。`waitpid`函数返回子进程的终止状态（`status`变量）。在此简单程序中，没有使用该值。如果需

11～12

要，可以用此值准确地判定子进程是因何终止的。

- 该程序的最主要限制是不能向所执行的命令传递参数，例如不能指定要列表的目录名，只能对工作目录执行ls命令。为了传递参数，先要分析输入行，然后用某种约定把参数分开（很可能使用空格或制表符），然后将分隔后的各个参数传递给execlp函数。尽管如此，此程序仍可用来说明UNIX系统的进程控制功能。

如果运行此程序，则得到下列结果。注意，该程序使用了一个不同的提示符（%），以区别于shell的提示符。

```
$ ./a.out
% date
Sun Aug  1 03:04:47 EDT 2004                    程序员挑灯夜战
% who
sar      :0        Jul 26 22:54
sar      pts/0     Jul 26 22:54 (:0)
sar      pts/1     Jul 26 22:54 (:0)
sar      pts/2     Jul 26 22:54 (:0)
% pwd
/home/sar/bk/apue/2e
% ls
Makefile
a.out
shell1.c
% ^D                                            键入文件结束符
$                                               常规的shell提示符
```

□

^D表示一个控制字符。控制字符是特殊字符，其形成方法是：在键盘上按下控制键——通常被标记为Control或Ctrl，同时按另一个键。Control-D或^D是默认的文件结束符。在第18章中讨论终端时，会介绍更多的控制字符。

4. 线程和线程ID

通常，一个进程只有一个控制线程（thread），同一时刻只执行一组机器指令。对于某些问题，如果不同部分各使用一个控制线程，那么整个问题解决起来就容易得多。另外，多个控制线程也能充分利用多处理器系统的并行性。

13

在一个进程内的所有线程共享同一地址空间、文件描述符、栈以及与进程相关的属性。因为它们能访问同一存储区，所以各线程在访问共享数据时需要采取同步措施以避免不一致性。

与进程相同，线程也用ID标识。但是，线程ID只在它所属进程内起作用。一个进程中的线程ID在另一个进程中并无意义。当在一进程中对多个线程进行操纵时，我们用线程ID引用相应的线程。

控制线程的函数与控制进程的函数类似，但另有一套。在进程模型建立很久之后，线程模型才被引入到UNIX系统中，这两个模型之间存在复杂的相互作用，在第12章中，我们会对此有所说明。

1.7　出错处理

当UNIX函数出错时，常常返回一个负值，而且整型变量errno通常被设置为含有附加信息的一个值。例如，open函数如成功执行则返回一个非负文件描述符，如出错则返回−1。在open出错时，有大约15种不同的errno值（文件不存在、权限问题等）。某些函数并不返回负

值而是使用另一种约定。例如，返回一个指向对象指针的大多数函数，在出错时，将返回一个null指针。

文件<errno.h>中定义了符号errno以及可以赋予它的各种常量，这些常量都以字符E开头。另外，UNIX系统手册第2部分的第1页intro(2)列出了所有这些出错常量。例如，若errno等于常量EACCES，这表示产生了权限问题（例如，没有打开所要求文件的足够权限）。

在Linux中，出错常量在errno(3)手册页中列出。

POSIX和ISO C将errno定义为这样一个符号，它扩展成为一个可修改的整型左值（lvalue）。这可以是包含出错编号的一个整数，或者是一个返回出错编号指针的函数。以前使用的定义是：

```
extern int errno;
```

但是在支持线程的环境中，多个线程共享进程地址空间，每个线程都有属于它自己的局部errno以避免一个线程干扰另一个线程。例如，Linux支持多线程存取errno，将其定义为：

```
extern int *__errno_location(void);
#define errno (*__errno_location())
```

对于errno应当知道两条规则。第一条规则是：如果没有出错，则其值不会被一个例程清除。因此，仅当函数的返回值指明出错时，才检验其值。第二条是：任一函数都不会将errno值设置为0，在<errno.h>中定义的所有常量都不为0。

14

C标准定义了两个函数，它们帮助打印出错信息。

```
#include <string.h>

char *strerror(int errnum);
```

返回值：指向消息字符串的指针

此函数将*errnum*（它通常就是errno值）映射为一个出错信息字符串，并且返回此字符串的指针。

perror函数基于errno的当前值，在标准出错上产生一条出错消息，然后返回。

```
#include <stdio.h>

void perror(const char *msg);
```

它首先输出由*msg*指向的字符串，然后是一个冒号，一个空格，接着是对应于errno值的出错信息，最后是一个换行符。

实 例

程序清单1-6显示了这两个出错函数的使用方法。

程序清单1-6　例示strerror和perror

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
```

```
        perror(argv[0]);
        exit(0);
    }
```

如果编译此程序，将结果送入文件a.out，然后执行它，则有：

```
$ ./a.out
EACCES: Permission denied
./a.out: No such file or directory
```

注意，我们将程序名（argv[0]，其值是./a.out）作为参数传递给perror。这是一个标准的UNIX惯例。使用这种方法，在程序作为管道线的一部分执行时，例如：

```
prog1 < inputfile | prog2 | prog3 > outputfile
```

15 我们就能分清三个程序中的哪一个产生了一条特定的出错消息。□

本书中的所有实例基本上都不直接调用strerror或perror，而是使用附录B中的出错函数。该附录中的出错函数使用了ISO C的可变参数表功能，用一条C语句就可处理出错情况。

出错恢复

可将在<errno.h>中定义的各种出错分成致命性的和非致命性的两类。对于致命性的错误，无法执行恢复动作，最多只能在用户屏幕上打印出一条出错消息，或者将一条出错消息写入日志文件中，然后终止。而对于非致命性的出错，有时可以较妥善地进行处理。大多数非致命性出错在本质上是暂时的，例如资源短缺，当系统中的活动较少时，这种出错很可能不会发生。

与资源相关的非致命性出错包括EAGAIN、ENFILE、ENOBUFS、ENOLCK、ENOSPC、ENOSR、EWOULDBLOCK，有时ENOMEM也是非致命性出错。当EBUSY指明共享资源正在使用时，也可将它作为非致命性出错处理。当EINTR中断一慢速系统调用时，可将它作为非致命性出错处理。在10.5节对此会作更多说明。

对于资源相关的非致命性出错，一般恢复动作是延迟一些时间，然后再试。这种技术可应用于其他情况。例如，假设一个出错表明一个网络连接不再起作用，那么应用程序可以在短时间延迟后重建该连接。某些应用使用指数补偿算法，在每次重复中等待更长时间。

最后，取决于应用程序的开发者，他可以决定那些出错是可恢复的。如若使用一种从错误中恢复的合理策略，那么由于避免了应用程序的异常终止，就能改善应用程序的健壮性。

1.8　用户标识

1. 用户ID

口令文件登录项中的*用户*ID（user ID）是个数值，它向系统标识各个不同的用户。系统管理员在确定一个用户的登录名的同时，确定其用户ID。用户不能更改其用户ID。通常每个用户有一个唯一的用户ID。下面将介绍内核如何使用用户ID检验该用户是否有执行某些操作的权限。

用户ID为0的用户为*根*（root）或*超级用户*（superuser）。在口令文件中，通常有一个登录项，其登录名为root，我们称这种用户的特权为超级用户特权。我们将在第4章中看到，如果一个进程具有超级用户特权，则大多数文件权限检查都不再进行。某些操作系统功能只限于向超级用户提供，超级用户对系统有自由的支配权。

Mac OS X客户端版本交由用户使用时，禁用超级用户账户，服务器版本则可使用该账户。在Apple的网站可以找到指导，说明如何才能启用该账户。见http://docs.info.apple.com/article_html?artnum=106290。

16

2. 组ID

口令文件登录项也包括用户的组ID（group ID），它是一个数值。组ID也是由系统管理员在指定用户登录名时分配的。一般来说，在口令文件中有多个记录项具有相同的组ID。组被用于将若干用户分到不同的项目组或部门中去。这种机制允许同组的各个成员之间共享资源（例如文件）。4.5节将说明可以设置文件的权限使组内所有成员都能存取该文件，而组外用户则不能。

组文件将组名映射为数字组ID，它通常是/etc/group。

对于权限，使用数值用户ID和数值组ID是历史上形成的。对于磁盘上的每个文件，文件系统都存放该文件所有者的用户ID和组ID。存放这两个值只需4个字节（假定每个都以双字节的整型值存放）。如果使用全长ASCII登录名和组名，则需较多的磁盘空间。另外，在查验权限期间，比较字符串较之比较整型数更消耗时间。

但是对于用户而言，使用名字比使用数值方便，所以口令文件包含了登录名和用户ID之间的映射关系，而组文件则包含了组名和组ID之间的映射关系。例如UNIX `ls -l`命令使用口令文件将数值用户ID映射为登录名，从而打印文件所有者的登录名。

早期的UNIX系统使用16位整型数表示用户ID和组ID。现今的UNIX系统使用32位整型数表示用户ID和组ID。

实 例

程序清单1-7用于打印用户ID和组ID。

程序清单1-7 打印用户ID和组ID

```
#include "apue.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

调用getuid和getgid以返回用户ID和组ID。运行该程序，将产生

```
$ ./a.out
uid = 205, gid = 105
```

17

3. 附加组ID

除了在口令文件中对一个登录名指定一个组ID外，大多数 UNIX系统版本还允许一个用户属于另外的组。这是从4.2BSD开始的，它允许一个用户属于多至16个另外的组。登录时，读文件/etc/group，寻找列有该用户作为其成员的前16个记录项就可得到该用户的附加组ID（supplementary group ID）。在下一章将说明，POSIX要求系统至少应支持8个附加组，实际上大多数系统至少支持16个附加组。

1.9 信号

信号（signal）是通知进程已发生某种情况的一种技术。例如，若某一进程执行除法操作，其除数为0，则将名为SIGFPE（浮点异常）的信号发送给该进程。进程如何处理信号有三种选择。

(1) 忽略该信号。有些信号表示硬件异常，例如，除以0或访问进程地址空间以外的单元等，因为这些异常产生的后果不确定，所以不推荐使用这种处理方式。

(2) 按系统默认方式处理。对于除以0的情况，系统默认方式是终止该进程。

(3) 提供一个函数，信号发生时则调用该函数，这被称为捕捉该信号。使用这种方式，我们只要提供自编的函数就将能知道什么时候产生了信号，并按所希望的方式处理它。

很多情况会产生信号。终端键盘上有两种产生信号的方法，分别称为**中断键**（interrupt key，通常是Delete键或Ctrl+C）和**退出键**（quit key，通常是Ctrl+\），它们被用于中断当前运行的进程。另一种产生信号的方法是调用名为kill的函数。在一个进程中调用此函数就可向另一个进程发送一个信号。当然这样做也有些限制：当向一个进程发送信号时，我们必须是该进程的所有者或者是超级用户。

回忆一下前面的简化shell程序（见程序清单1-5）。如果调用此程序，然后键入中断键，则执行此程序的进程终止。产生这种后果的原因是，对于此信号（SIGINT）的系统默认动作是终止进程。该进程没有告诉系统内核对此信号作何种处理，所以系统按默认方式终止该进程。

为使该程序能捕捉到此信号，它需要调用signal函数，由其指定当产生SIGINT信号时要调用的函数名。为此编写了名为sig_int的函数，当其被调用时，它只是打印一条消息，然后打印一个新提示符。在程序清单1-5中添加了11行，构成了程序清单1-8（添加的11行以行首的+号表示）。

18

程序清单1-8　从标准输入读命令并执行

```
  #include "apue.h"
  #include <sys/wait.h>

+ static void sig_int(int);       /* our signal-catching function */
+
  int
  main(void)
  {
      char    buf[MAXLINE];    /* from apue.h */
      pid_t   pid;
      int     status;

+     if (signal(SIGINT, sig_int) == SIG_ERR)
+         err_sys("signal error");
+
      printf("%% ");  /* print prompt (printf requires %% to print %) */
      while (fgets(buf, MAXLINE, stdin) != NULL) {
          if (buf[strlen(buf) - 1] == '\n')
              buf[strlen(buf) - 1] = 0; /* replace newline with null */

          if ((pid = fork()) < 0) {
```

```
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
+
+ void
+ sig_int(int signo)
+ {
+     printf("interrupt\n%% ");
+ }
```

因为大多数重要的应用程序都将使用信号，所以第10章将详细介绍信号。 □ [19]

1.10 时间值

长期以来，UNIX系统一直使用两种不同的时间值：

(1) 日历时间。该值是自1970年1月1日00:00:00以来国际标准时间（UTC）所经过的秒数累计值（早期的手册称UTC为格林尼治标准时间）。这些时间值可用于记录文件最近一次的修改时间等。

系统基本数据类型time_t用于保存这种时间值。

(2) 进程时间。这也被称为CPU时间，用以度量进程使用的中央处理机资源。进程时间以时钟滴答计算，历史上曾经取每秒钟为50、60或100个滴答。

系统基本数据类型clock_t用于保存这种时间值。2.5.4节将说明如何用sysconf函数得到每秒时钟滴答数。

当度量一个进程的执行时间时（见3.9节），UNIX系统使用三个进程时间值：

- 时钟时间。
- 用户CPU时间。
- 系统CPU时间。

时钟时间又称为*墙上时钟时间*（wall clock time）。它是进程运行的时间总量，其值与系统中同时运行的进程数有关。每当在本书中述及时钟时间时，都是在系统中没有其他活动时进行度量的。

用户CPU时间是执行用户指令所用的时间。系统CPU时间是为该进程执行内核程序所经历的时间。例如，每当一个进程执行一个系统服务时，例如read或write，则在内核内执行该服务所花费的时间就计入该进程的系统CPU时间。用户CPU时间和系统CPU时间之和常被称为CPU时间。

要取得任一进程的时钟时间、用户时间和系统时间是很容易的——只要执行命令time(1)，其参数是要度量其执行时间的命令，例如：

```
$ cd /usr/include
$ time -p grep _POSIX_SOURCE */*.h > /dev/null

real    0m0.81s
user    0m0.11s
sys     0m0.07s
```

time命令的输出格式与所使用的shell有关，其原因是某些shell并不运行/usr/bin/time，而是使用一内置函数测量命令运行所使用的时间。

20

8.16节将说明一个运行进程如何取得这三个时间。关于时间和日期的一般说明见6.10节。

1.11 系统调用和库函数

所有的操作系统都提供多种服务的入口点，程序由此向内核请求服务。各种版本的UNIX实现都提供定义明确、数量有限、可直接进入内核的入口点，这些入口点被称为*系统调用*（见图1-1）。Research UNIX第7版提供了约50个系统调用，4.4BSD提供了约110个，而SVR4则提供了约120个。Linux的不同版本提供了240～260个系统调用。FreeBSD大约提供了320个。

系统调用接口总是在《UNIX程序员手册》的第2部分中说明，其定义也包括在C语言中，这与具体系统如何调用系统调用的实现技术无关。与很多早期的操作系统不同，这些系统按传统方式在机器的汇编语言中定义内核入口点。

UNIX所使用的技术是为每个系统调用在标准C库中设置一个具有同样名字的函数。用户进程用标准C调用序列来调用这些函数，然后，函数又用系统所要求的技术调用相应的内核服务。例如函数可将一个或多个C参数送入通用寄存器，然后执行某个产生软中断进入内核的机器指令。从应用角度考虑，可将系统调用视作为C函数。

《UNIX程序员手册》的第3部分定义了程序员可以使用的通用函数。虽然这些函数可能会调用一个或多个内核的系统调用，但是它们并不是内核的入口点。例如，printf函数会调用write系统调用以输出一个字符串，但函数strcpy（复制一字符串）和atoi（变换ASCII为整数）并不使用任何系统调用。

从实现者的角度观察，系统调用和库函数之间有重大区别；但从用户角度来看，其区别并不非常重要。在本书中，系统调用和库函数都以C函数的形式出现，两者都为应用程序提供服务。但是，我们应当理解，必要时我们可以替换库函数，而通常却不能替换系统调用。

以存储器分配函数malloc为例。有多种方法可以进行存储器分配及与其相关的无用区收集操作（最佳适应、首次适应等），并不存在对所有程序都最佳的一种技术。UNIX系统调用中处理存储器分配的是sbrk(2)，它不是一个通用的存储器管理器。它按指定字节数增加或减少进程地址空间。如何管理该地址空间却取决于进程。存储器分配函数malloc(3)实现一种特定类型的分配。如果我们不喜欢其操作方式，则可以定义自己的malloc函数，它很可能将使用sbrk系统调用。事实上，有很多软件包，它们使用sbrk系统调用实现自己的存储器分配算法。图1-2显示了应用程序、malloc函数以及sbrk系统调用之间的关系。

从中可见，两者职责不同，内核中的系统调用分配另外一块空间给进程，而库函数malloc则在用户层次管理这一空间。

21

另一个可说明系统调用和库函数之间的差别的例子是UNIX系统提供的决定当前时间和日期的接口。某些操作系统提供一个系统调用以返回时间，提供另一个以返回日期。任何特殊的处理，例如正常时制和夏时制之间的转换，则由内核处理或要求人为干预。UNIX系统则不同，

它只提供一条系统调用，该系统调用返回国际标准时间1970年1月1日零点以来所经过的秒数。对该值的任何解释，例如将其变换成人们可读的、适用于本地时区的时间和日期，都留给用户进程进行处理。在标准C库中，提供了若干例程以处理大多数情况。这些库函数处理各种细节，例如各种夏时制算法等。

应用程序可以调用系统调用或者库函数，而很多库函数则会调用系统调用。这在图1-3中显示。

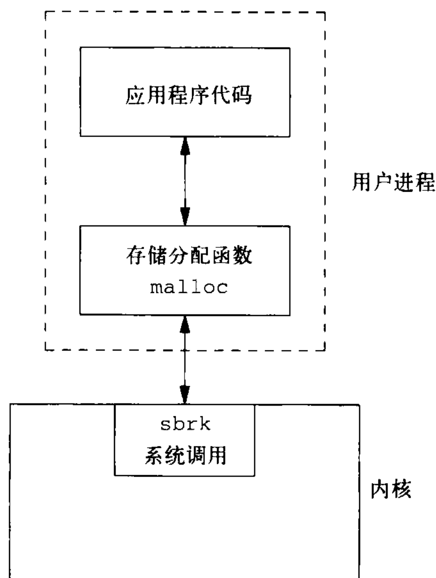


图1-2 malloc函数和sbrk系统调用

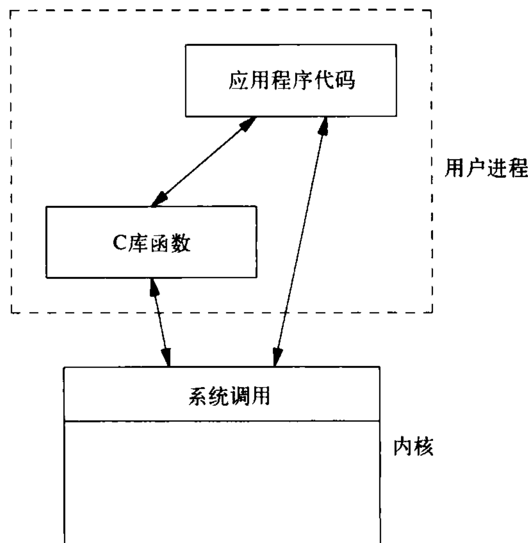


图1-3 C库函数和系统调用之间的差别

系统调用和库函数之间的另一个差别是：系统调用通常提供一种最小接口，而库函数通常提供比较复杂的功能。我们从sbrk系统调用和malloc库函数之间的差别中可以看到这一点，在以后比较不带缓冲的I/O函数（见第3章）以及标准I/O函数（见第5章）时，还将看到这种差别。

进程控制系统调用（fork、exec和wait）通常由用户应用程序直接调用（请回忆程序清单1-5中的基本shell）。但是为了简化某些常见的情况，UNIX系统也提供了一些库函数，例如system和popen。8.13节将说明system函数的一种实现，它使用基本的进程控制系统调用。10.18节还将强化这一实例以正确地处理信号。 22

为使读者了解大多数程序员应用的UNIX系统接口，我们不得不既说明系统调用，又介绍某些库函数。例如，若只说明sbrk系统调用，那么就会忽略很多应用程序使用的malloc库函数。除了必须要区分两者时，本书都将使用术语*函数*（function）来表示系统调用和库函数两者。

1.12 小结

本章快速浏览了UNIX系统。说明了某些以后会多次用到的基本术语，介绍了一些小的UNIX程序实例，使读者对本书其余部分将会进一步介绍的内容有一些感性认识。

下一章将讲述UNIX系统的标准，以及这方面的工作对当前系统的影响。标准，特别是ISO C标准和POSIX.1标准将影响本书的余下部分。 23

习题

1.1 在系统上查证，除根目录外，目录 . 和 .. 是不同的。

1.2 分析程序清单1-4的输出，说明进程ID为852和853的进程发生了什么情况。

1.3 在1.7节中，`perror`的参数是用ISO C的属性`const`定义的，而`strerror`的整型参数则没有用此属性定义，为什么？

1.4 在附录B包含了出错处理函数`err_sys`，当调用该函数时，先保存了`errno`的值，为什么？

1.5 若日历时间存放在带符号的32位整型数中，那么到哪一年它将溢出？可以用什么方法扩展浮点数？它们是否与已存在的应用相兼容？

1.6 若进程时间存放在带符号的32位整型数中，而且每秒为100滴答，那么经过多少天后该时间值将会溢出？

24

第2章

UNIX标准化及实现

2.1 引言

在UNIX编程环境和C程序设计语言的标准化方面已经做了很多工作。虽然UNIX应用程序在不同的UNIX操作系统版本之间进行移植相当容易，但是20世纪80年代UNIX版本的剧增以及它们之间差别的扩大，导致很多大用户（例如美国政府）呼吁对其进行标准化。

本章首先将介绍过去20年来进行的各种标准化工作，然后讨论这些UNIX编程标准对本书所列举的各种UNIX操作系统实现的影响。所有标准化工作的一个重要部分是对每种实现必须定义的各种限制进行说明，所以我们将说明这些限制以及确定其值的各种方法。

2.2 UNIX标准化

2.2.1 ISO C

1989年下半年，C程序设计语言的ANSI标准X3.159-1989得到批准。此标准已被采纳为国际标准ISO/IEC 9899: 1990。ANSI是美国国家标准学会（American National Standards Institute），它在国际标准化组织（International Organization for Standardization，ISO）中是代表美国的成员。IEC是国际电子技术委员会（International Electrotechnical Commission）的缩写。

ISO C标准现在由ISO/IEC的C程序设计语言国际标准化工作组维护和开发，该工作组被称为ISO/IEC JTC1/SC22/WG14，简称WG14。ISO C标准的意图是提供C程序的可移植性，使其能适合于大量不同的操作系统，而不只是UNIX系统。此标准不仅定义了C程序设计语言的语法和语义，还定义了其标准库[ISO 1999第7章；Plauger 1992；Kernighan及Ritchie 1988中的附录B]。因为所有现今的UNIX系统（例如本书介绍的几个UNIX系统）都提供C标准中定义的库例程，所以该标准库是很重要的。

在1999年，ISO C标准被更新为ISO/IEC 9899：1999。新标准显著改善了对进行数值处理的应用程序的支持。除了对某些函数原型增加了关键字`restrict`外，这些改变并不影响本书中说明的POSIX标准。该关键字用于告诉编译器，哪些指针引用是可以优化的，其方法是指明指针指向的对象，在函数中只通过该指针进行访问。

如同大多数标准一样，在批准标准和修改软件以使其符合标准这两者之间有一段时间上的延迟。随着供应商的编译系统不断演进，对ISO C标准最新版本的支持也越来越多。

> gcc对ISO C标准1999版本的当前符合程度的总结可见：`http://www.gnu.org/software/gcc/c99status.html`。

按照该标准定义的各个头文件（header），可将ISO C库分成24个区。表2-1中列出了C标准定义的各个头文件。POSIX.1标准包括这些头文件以及另外一些头文件。表中也列出了四种UNIX实现（FreeBSD 5.2.1、Linux 2.4.22、Mac OS X 10.3和Solaris 9）所支持的头文件。本章后面将对这四种UNIX实现进行说明。

表2-1　ISO C标准定义的头文件

头　文　件	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说　　明
<assert.h>	•	•	•	•	验证程序断言
<complex.h>	•	•	•		支持复数算术运算
<ctype.h>	•	•	•	•	字符类型
<errno.h>	•	•	•	•	出错码（1.7节）
<fenv.h>		•	•		浮点环境
<float.h>	•	•	•	•	浮点常量
<inttypes.h>	•	•	•	•	整型格式转换
<iso646.h>	•	•	•	•	替代关系操作符宏
<limits.h>	•	•	•	•	实现常量（2.5节）
<locale.h>	•	•	•	•	局部类别
<math.h>	•	•	•	•	数学常量
<setjmp.h>	•	•	•	•	非局部goto（7.10节）
<signal.h>	•	•	•	•	信号(第10章)
<stdarg.h>	•	•	•	•	可变参数表
<stdbool.h>	•	•	•	•	布尔类型和值
<stddef.h>	•	•	•	•	标准定义
<stdint.h>	•	•	•		整型
<stdio.h>	•	•	•	•	标准I/O库(第 5章)
<stdlib.h>	•	•	•	•	实用程序函数
<string.h>	•	•	•	•	字符串操作
<tgmath.h>		•			通用类型数学宏
<time.h>	•	•	•	•	时间和日期(6.10节)
<wchar.h>	•	•	•	•	扩展的多字节和宽字符支持
<wctype.h>	•	•	•	•	宽字符分类和映射支持

ISO C头文件依赖于操作系统所配置的C编译器的版本。在考虑表2-1时，应当注意FreeBSD 5.2.1配置了gcc 3.3.3版，Solaris 9同时配置了gcc 2.95.3版和gcc 3.2版，Mandrake 9.2（Linux 2.4.22）配置了gcc 3.3.1版，Mac OS X配置了gcc 3.3版。Mac OS X还包括了gcc的较早版本。

2.2.2　IEEE POSIX

POSIX是一系列由IEEE（Institute of Electrical and Electronics Engineers，电气与电子工程师协会）制定的标准。POSIX指的是可移植的操作系统接口（Portable Operating System Interface）。它原来指的只是IEEE标准1003.1-1988（操作系统接口），后来则扩展成包括很多标记为1003的标准及标准草案，包括shell和实用程序（1003.2）。

与本书相关的是1003.1操作系统接口标准，该标准的目的是提高应用程序在各种UNIX系统环境之间的可移植性。它定义了“依从POSIX的”（POSIX compliant）操作系统必须提供的各

种服务。该标准已被大多数计算机制造商采用。虽然1003.1标准是以UNIX操作系统为基础的，但是它并不限于UNIX和类似于UNIX的系统。确实，有些供应专有操作系统的制造商也声称这些系统将依从POSIX（同时还保有它们的所有专有功能）。 26

由于1003.1标准定义了一个**接口**（interface）而不是一种**实现**（implementation），所以并不区分系统调用和库函数。标准中的所有例程都称为函数。

标准是不断演变的，1003.1标准也不例外。该标准的1988版，即IEEE 1003.1-1988，经修改后提交给ISO。它没有增加新的接口或功能，但修订了文本。最终的文档作为IEEE Std.1003.1-1990正式出版[IEEE 1990]，这也就是国际标准ISO/IEC 9945-1:1990。该标准通常被称为POSIX.1，本书将使用此标准。

IEEE 1003.1工作组继续对标准做出修改，并在1993年出版了IEEE 1003.1标准的修订版。它包括了1003.1-1990标准和1003.1b-1993实时扩展标准。1996年，该标准再次更新为国际标准ISO/IEC 9945-1:1996。它包括了多线程编程的接口，称为pthreads，指的就是POSIX线程。1999年出版了IEEE标准1003.1d-1999，其中增加了更多实时接口。一年后，出版了IEEE标准1003.1j-2000和1003.1q-2000，前者包含了更多实时接口，后者增加了标准在事件跟踪方面的扩展。 27

1003.1的2001版与以前各版本有较大的差别，它组合了1003.1的多次修订、1003.2标准以及Single UNIX Specification（SUS）第2版的若干部分（对于SUS，后面将作更多说明）。最终形成了IEEE标准1003.1-2001，其中包括了下列几个标准。

- ISO/IEC 9945-1（IEEE标准1003.1-1996），它包括
    - IEEE标准1003.1 - 1990。
    - IEEE标准1003.1b - 1993（实时扩展）。
    - IEEE标准1003.1c- 1995（pthreads）。
    - IEEE标准1003.1i- 1995（实时技术勘误表）。
- IEEE P1003.1a标准草案（系统接口修订版）。
- IEEE标准1003.1d - 1999（高级实时扩展）。
- IEEE标准1003.1j - 2000（更高级的实时扩展）。
- IEEE标准1003.1q - 2000（文件跟踪）。
- IEEE标准1003.2d - 1994（批处理扩展）。
- IEEE P1003.2b草案标准（附加的实用程序）。
- IEEE标准 1003.1g - 2000（协议无关接口）的某些部分。
- ISO/IEC 9945-2（IEEE标准1003.2 - 1993）。
- Single UNIX Specification第2版的基本规范，包括
    - 系统接口定义，第5发行版。
    - 命令和实用程序，第5发行版。
    - 系统接口和头文件，第5发行版。
- 开放组技术标准，网络服务，5.2 发行版。
- ISO/IEC 9899:1999，C编程语言。

表2-2、表2-3以及表2-4总结了POSIX.1指定的必需和可选的头文件。因为POSIX.1包括ISO C标准库函数，所以它还需要表2-1中列出的头文件。这4个表总结了本书所讨论的4种UNIX系统实现中包括的头文件。

表2-2　POSIX标准定义的必需的头文件

头文件	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说　明
<dirent.h>	•	•	•	•	目录项（4.21节）
<fcntl.h>	•	•	•	•	文件控制（3.14节）
<fnmatch.h>	•	•	•	•	文件名匹配类型
<glob.h>	•	•	•	•	路径名模式匹配类型
<grp.h>	•	•	•	•	组文件（6.4节）
<netdb.h>	•	•	•	•	网络数据库操作
<pwd.h>	•	•	•	•	口令文件（6.2节）
<regex.h>	•	•	•	•	正则表达式
<tar.h>	•	•	•	•	tar归档值
<termios.h>	•	•	•	•	终端I/O（第18章）
<unistd.h>	•	•	•	•	符号常量
<utime.h>	•	•	•	•	文件时间（4.19节）
<wordexp.h>	•	•		•	字扩展类型
<arpa/inet.h>	•	•	•	•	Internet定义（第16章）
<net/if.h>	•	•	•	•	套接字本地接口（第16章）
<netinet/in.h>	•	•	•	•	Internet地址族（16.3节）
<netinet/tcp.h>	•	•	•	•	传输控制协议定义
<sys/mman.h>	•	•	•	•	内存管理声明
<sys/select.h>	•	•	•	•	select函数（14.5.1节）
<sys/socket.h>	•	•	•	•	套接字接口（第16章）
<sys/stat.h>	•	•	•	•	文件状态（第4章）
<sys/times.h>	•	•	•	•	进程时间（8.16节）
<sys/types.h>	•	•	•	•	基本系统数据类型（2.8节）
<sys/un.h>	•	•	•	•	UNIX域套接字定义（17.3节）
<sys/utsname.h>	•	•	•	•	系统名（6.9节）
<sys/wait.h>	•	•	•	•	进程控制（8.6节）

表2-3　POSIX标准定义的XSI扩展头文件

头文件	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说　明
<cpio.h>	•	•		•	cpio归档值
<dlfcn.h>	•	•	•	•	动态链接
<fmtmsg.h>	•	•		•	消息显示结构
<ftw.h>		•		•	文件树漫游（4.21节）
<iconv.h>		•	•	•	代码集转换实用程序
<langinfo.h>	•	•	•	•	语言信息常量
<libgen.h>	•	•	•	•	模式匹配函数定义
<monetary.h>	•	•	•	•	货币类型
<ndbm.h>	•		•	•	数据库操作
<nl_types.h>	•	•	•	•	消息类别
<poll.h>	•	•	•	•	轮询函数（14.5.2节）
<search.h>	•	•	•	•	搜索表
<strings.h>	•	•	•	•	字符串操作

（续）

头 文 件	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说 明
<syslog.h>	•	•	•	•	系统出错日志记录（13.4节）
<ucontext.h>	•	•	•	•	用户上下文
<ulimit.h>	•	•	•	•	用户限制
<utmpx.h>		•		•	用户账户数据库
<sys/ipc.h>	•	•	•	•	IPC（15.6节）
<sys/msg.h>	•	•		•	消息队列（15.7节）
<sys/resource.h>	•	•	•	•	资源操作（7.11节）
<sys/sem.h>	•	•	•	•	信号量（15.9节）
<sys/shm.h>	•	•	•	•	共享存储（15.9节）
<sys/statvfs.h>	•	•		•	文件系统信息
<sys/time.h>	•	•	•	•	时间类型
<sys/timeb.h>	•	•	•	•	附加的日期和时间定义
<sys/uio.h>	•	•	•	•	矢量I/O操作（14.7节）

表2-4 POSIX标准定义的可选头文件

头 文 件	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说 明
<aio.h>	•	•	•	•	异步I/O
<mqueue.h>	•			•	消息队列
<pthread.h>	•	•	•	•	线程（第11、12章）
<sched.h>	•	•	•	•	执行调度
<semaphore.h>	•	•	•	•	信号量
<spawn.h>		•			实时spawn接口
<stropts.h>		•		•	XSI STREAMS接口（14.4节）
<trace.h>					事件跟踪

本书中描述了POSIX.1的2001版，包括ISO C标准所指定的各个函数。其接口分成了两类：必需接口和可选接口。可选接口按功能又进一步分成50个区。表2-5中按它们各自的选项代码总结了没有被弃用的编程接口。选项代码是由2～3个字符构成的字母缩写，以便标识属于各个功能区的接口。选项代码会突出显示手册相关页面上的文本，表明接口依赖于对特定选项的支持。很多选项处理实时扩展。

表2-5 POSIX.1可选接口组和代码

代 码	SUS强制要求	符 号 常 量	说 明
ADV		_POSIX_ADVISORY_INFO	建议性信息（实时）
AIO		_POSIX_ASYNCHRONOUS_IO	异步输入和输出（实时）
BAR		_POSIX_BARRIERS	屏障（实时）
CPT		_POSIX_CPUTIME	进程CPU时钟（实时）
CS		_POSIX_CLOCK_SELECTION	时钟选择（实时）
CX	•		ISO C标准扩展
FSC	•	_POSIX_FSYNC	文件同步
IP6		_POSIX_IPV6	IPv6接口

（续）

代　码	SUS强制要求	符号常量	说　　明
MF	•	_POSIX_MAPPED_FILES	存储映射的文件
ML		_POSIX_MEMLOCK	进程存储区加锁（实时）
MLR		_POSIX_MEMLOCK_RANGE	存储区域加锁（实时）
MON		_POSIX_MONOTONIC_CLOCK	单调时钟（实时）
MPR	•	_POSIX_MEMORY_PROTECTION	存储保护
MSG		_POSIX_MESSAGE_PASSING	消息传送（实时）
MX			IEC 60559浮点选项
PIO		_POSIX_PRIORITIZED_IO	优先输入和输出
PS		_POSIX_PRIORITIZED_SCHEDULING	进程调度（实时）
RS		_POSIX_RAW_SOCKETS	原始套接字
RTS		_POSIX_REALTIME_SIGNALS	实时信号扩展
SEM		_POSIX_SEMAPHORES	信号量（实时）
SHM		_POSIX_SHARED_MEMORY_OBJECTS	共享存储对象（实时）
SIO		_POSIX_SYNCHRONIZED_IO	同步输入和输出（实时）
SPI		_POSIX_SPIN_LOCKS	自旋锁（实时）
SPN		_POSIX_SPAWN	产生（实时）
SS		_POSIX_SPORADIC_SERVER	进程散发性服务器（实时）
TCT		_POSIX_THREAD_CPUTIME	线程CPU时钟（实时）
TEF		_POSIX_TRACE_EVENT_FILTER	跟踪事件过滤器
THR	•	_POSIX_THREADS	线程
TMO		_POSIX_TIMEOUTS	超时（实时）
TMR		_POSIX_TIMERS	计时器（实时）
TPI		_POSIX_THREAD_PRIO_INHERIT	线程优先级继承（实时）
TPP		_POSIX_THREAD_PRIO_PROTECT	线程优先级保护（实时）
TPS		_POSIX_THREAD_PRIORITY_SCHEDULING	线程执行调度（实时）
TRC		_POSIX_TRACE	跟踪
TRI		_POSIX_TRACE_INHERIT	跟踪继承
TRL		_POSIX_TRACE_LOG	跟踪日志
TSA	•	_POSIX_THREAD_ATTR_STACKADDR	线程栈地址属性
TSF	•	_POSIX_THREAD_SAFE_FUNCTIONS	线程安全的函数
TSH	•	_POSIX_THREAD_PROCESS_SHARED	线程进程共享的同步
TSP		_POSIX_THREAD_SPORADIC_SERVER	线程散发性服务器（实时）
TSS	•	_POSIX_THREAD_ATTR_STACKSIZE	线程栈地址大小
TYM		_POSIX_TYPED_MEMORY_OBJECTS	类型化的存储对象（实时）
XSI	•	_XOPEN_UNIX	X/Open扩展接口
XSR		_XOPEN_STREAMS	XSI STREAMS

POSIX.1没有包括超级用户（superuser）这样的概念，代之以规定某些操作要求"适当的特权"，POSIX.1将此术语的定义留由具体实现进行解释。某些符合美国国防部安全性指南的UNIX系统具有很多不同的安全级。本书仍使用传统的UNIX术语，并指明要求超级用户特权的操作。

经过近20年的工作，相关标准已经成熟稳定。POSIX.1标准现由称为Austin Group（http://www.opengroup.org/austin）的开放工作组维护。为了保证它们与实际需求吻合，仍需经常对这些标准进行更新或再修订。

2.2.3 Single UNIX Specification

Single UNIX Specification（单一UNIX规范）是POSIX.1标准的一个超集，定义了一些附加的接口，这些接口扩展了基本的POSIX.1规范所提供的功能。相应的系统接口全集被称为X/Open系统接口（XSI, X/Open System Interface）。_XOPEN_UNIX符号常量标识了（相对于基本POSIX.1接口而言）XSI扩展的接口。

XSI还定义了实现必须支持POSIX.1的哪些可选部分才能认为是遵循XSI（XSI conforming）的。它们包括文件同步、存储映射文件、存储保护及线程接口，并在表2-5中标明是“SUS强制要求”。只有遵循XSI的实现才能称为UNIX系统。

> Open Group拥有UNIX商标，并且使用Single UNIX Specification来定义一个实现必须支持的接口，这样的实现才能称为UNIX系统。实现必须以文件形式提供符合性声明，并通过验证符合性的测试包测试，以及得到使用UNIX商标的许可。

28 ~ 31

在XSI中定义的某些附加的接口是必需的，其他的则是可选的。依据常用的功能，接口被分成如下若干选项组（option group）：

- 加密：由符号常量_XOPEN_CRYPE标记。
- 实时：由符号常量_XOPEN_REALTIME标记。
- 高级实时。
- 实时线程：由符号常量_XOPEN_REALTIME_THREADS标记。
- 高级实时线程。
- 跟踪。
- XSI STREAMS（流）：由符号常量_XOPEN_STREAMS标记。
- 遗留接口：由符号常量_XOPEN_LEGACY标记。

Single UNIX Specification（SUS）由Open Group发布，Open Group成立于1996年，由两个业界社团X/Open和Open Software Foundation（OSF）合并而成。X/Open过去一直在出版X/Open Portability Guide（X/Open可移植性指南），它采用了若干特定标准，填补了其他标准缺失功能的空白。这些指南的目的是改善应用程序的可移植性，使其不仅仅符合已出版的标准。

Single UNIX Specification的第一个版本由X/Open 在1994年出版，因为它大约包含了1170个接口，因此也被称为“Spec 1170”。它源自Common Open Software Environment（COSE）的倡议，该倡议的目标是进一步改善应用程序在所有UNIX操作系统实现之间的可移植性。COSE组的成员包括Sun、IBM、HP、Novell/USL以及OSF，该组织较之仅仅支持标准前进了一大步。此外，他们调查了若干常见的商业应用程序所使用的接口，并从中选出了1170个接口。这些接口也包括在下列各标准中：X/Open Common Application Environment（CAE）第4发行版（称为XPG4，以表示它与其前身X/Open Portability Guide的历史关系）、System V Interface Definition（SVID，系统V接口定义）第3版、Level 1接口、OSF Application Environment Specification（AES）Full Use 接口。

Single UNIX Specification第2版由Open Group在1997年出版。新版本增加了对一些功能的支持，包括线程、实时接口、64位处理、大文件以及增强的多字节字符处理等。

Single UNIX Specification第3版（简写为SUS v3）由Open Group在2001年出版。SUS v3的基本规范与IEEE标准1003.1-2001相同，分成4部分：基本定义、系统接口、Shell和实用程序以

及基本理论。SUS v3还包括X/Open Curses Issue 4第2版，但该规范并不是POSIX.1的组成部分。

2002年，ISO将该版本批准为国际标准ISO/IEC 9945:2002。Open Group在2003年再次更新了1003.1标准，使其包括了技术方面的更正，ISO将其批准为国际标准ISO/IEC 9945:2003。2004年4月，Open Group出版了Single UNIX Specification第3版，2004版本。它包括了对标准主要正文更多在技术上的更正。

32

2.2.4 FIPS

FIPS的含义是联邦信息处理标准（Federal Information Processing Standard）。它由美国政府出版，用于计算机系统的采购。FIPS 151-1（1989年4月）基于IEEE标准1003.1-1988及ANSI C标准草案。此后是FIPS 151-2（1993年5月），它基于IEEE标准1003.1-1990。某些在POSIX.1中列为可选的功能，在FIPS 151-2中是必需的。所有这些可选功能在POSIX.1-2001中已成为强制性要求。

POSIX.1 FIPS的影响是：它要求任何希望向美国政府销售POSIX.1兼容的计算机系统的厂商应支持POSIX.1的某些可选功能。因为POSIX.1 FIPS的影响正逐步减退，所以在本书中我们将不再进一步考虑它。

2.3 UNIX系统实现

上一节描述了由各自独立的组织所制定的三个标准：ISO C、IEEE POSIX以及Single UNIX Specification。但是，标准只是接口的规范。这些标准是如何与现实世界相关联的呢？这些标准由制造商采用，然后转变成具体实现。本书中我们感兴趣的是这些标准和它们的具体实现。

在McKusick等[1996]的1.1节中给出了UNIX系统家族树的详细历史（还有很好的一幅图片）。UNIX的各种版本和变体都起源于在PDP-11系统上运行的UNIX分时系统第6版（1976年）和第7版（1979年）（通常称为V6和V7）。这两个版本是在贝尔实验室以外首先得到广泛应用的UNIX系统。从树上演变出三个分支：

(1) AT&T分支，从此导出了系统Ⅲ和系统V（被称为UNIX的商用版本）。

(2) 加州大学伯克利分校分支，从此导出4.xBSD实现。

(3) 由AT&T贝尔实验室的计算科学研究中心开发的UNIX研究版本，从此导出UNIX分时系统第8、第9版以及于1990年发布的最后一版第10版。

2.3.1 SVR4

SVR4（UNIX System V Release 4，UNIX系统V第4版)是AT&T的UNIX系统实验室（USL，其前身是AT&T的UNIX Software Operation）的产品。它将下列系统的功能合并到一个一致的操作系统中：AT&T的UNIX系统V第3.2版（SVR3.2）、Sun Microsystems公司的SunOS操作系统、加州大学伯克利分校的4.3BSD版本以及微软的Xenix系统（Xenix是在V7的基础上开发的，后来又采用了很多系统V的功能）。其源代码于1989年后期发布，并在1990年开始向终端用户提供。SVR4符合POSIX 1003.1标准和X/Open Portability Guide第3版（XPG3）标准。

33

AT&T也出版了系统V接口定义（SVID）[AT&T 1989]。SVID第3版说明了UNIX系统要达到SVR4质量要求所应提供的功能。如同POSIX.1一样，SVID定义了一个接口，而不是一种实现。SVID并不区分系统调用和库函数。对于一个SVR4的具体实现，应查看其参考手册，以了

解系统调用和库函数的不同之处[AT&T 1990e]。

2.3.2 4.4BSD

BSD（Berkeley Software Distribution）版是由加州大学伯克利分校的计算机系统研究组（CSRG）研究开发和分发的。4.2BSD于1983年问世，4.3BSD则于1986年发布。这两个版本都在VAX小型机上运行。它们的下一个版本4.3BSD Tahoe于1988年发布，在一台称为Tahoe的小型机上运行（Leffler等[1989]说明了4.3BSD Tahoe版）。其后又有1990年的4.3BSD Reno版，它支持很多POSIX.1的功能。

最初的BSD系统包含了AT&T专有的源代码，并需要AT&T许可证。为了获得BSD系统的源代码，首先需要持有AT&T的UNIX源代码许可证。这种情况正在得到改变，近几年来，愈来愈多的AT&T源代码被替换成非AT&T源代码，很多加到BSD系统上的新功能也来自于非AT&T方面。

1989年，伯克利分校的计算机系统研究组将4.3BSD Tahoe中很多非AT&T源代码包装成BSD网络软件1.0版，并使其成为可公开得到的软件。其后则有BSD网络软件2.0版（1991），它是从4.3BSD Reno版派生出来的，其目的是使大部分（如果不是全部的话）4.4BSD系统不再受AT&T许可证的限制，这样，大家都可以得到源代码。

4.4BSD-Lite是CSRG期望开发的最后一个版本。由于与USL产生了法律纠纷，该版本曾一度延迟推出。在纠纷解决后，4.4BSD-Lite立即于1994年发布，并且不再需要UNIX源代码使用许可证就可以使用它。1995年CSRG发布排除了故障的版本。4.4BSD-Lite第2版是CSRG的最后一个BSD版本（McKusick等[1996]描述了该BSD版本）。

在伯克利所进行的UNIX开发工作是从PDP-11开始的，然后转移到VAX小型机上，接着又转移到工作站上。20世纪90年代早期，伯克利得到支持，在广泛应用的80386个人计算机上开发BSD版本，结果产生了386BSD。这一工作是由Bill Jolitz完成的。其相关文档发表在1991年*Dr.Dobb Journal*上（每月一篇的系列文章）。其中很多代码出现在BSD网络软件2.0版中。 34

2.3.3 FreeBSD

FreeBSD的基础是4.4BSD-Lite操作系统。在加州大学伯克利分校的CSRG决定终止其在UNIX操作系统的BSD版本上的研发工作后，并且386BSD项目看起来似乎被忽视了太长的时间，为了继续坚持BSD系列，设立了FreeBSD项目。

由FreeBSD项目产生的所有软件（包括其二进制代码和源代码）都是免费使用的。为了测试本书中的实例采用了4个系统，FreeBSD 5.2.1操作系统是其中的一个。

> 还有其他几个基于BSD的免费操作系统。NetBSD项目（http://www.netbsd.org）类似于FreeBSD项目，但是更注重不同硬件平台之间的可移植性。OpenBSD项目（http://www.openbsd.org）也类似于FreeBSD项目，但更注重安全性。

2.3.4 Linux

Linux是一种提供丰富的UNIX编程环境的操作系统，在GNU公用许可证指导下，Linux是免费使用的。Linux的普及是计算机产业中一道亮丽的风景线。Linux经常是支持新硬件的第一个操作系统，这一点使其引人注目。

Linux是由Linus Torvalds在1991年为替代MINIX而研发的。一位当时名不见经传人物的努力掀起了澎湃巨浪，吸引了遍布全世界的很多软件开发者，自愿地贡献出他们大量的时间来使用和不断地增强Linux。

Linux的Mandrake 9.2分发版是用以测试本书实例的操作系统之一。该系统使用了Linux操作系统内核2.4.22版。

2.3.5 Mac OS X

与其以前的版本相比，Mac OS X使用了完全不同的技术。其核心操作系统被称为Darwin，它基于Mach内核（Accetta等[1986]）和FreeBSD操作系统的组合。类似于FreeBSD和Linux，Darwin是一个开放源代码项目。

Mac OS X 10.3版（Darwin 7.4.0）是用以测试本书实例的操作系统之一。

2.3.6 Solaris

Solaris是由Sun公司开发的UNIX系统版本。它基于SVR4，并在10余年间由Sun公司的工程师对其进行了不断的增强。它是唯一在商业上取得成功的SVR4后裔，并被正式验证为UNIX系统。（关于UNIX验证的更多信息，请参见`http://www.opengroup.org/ certification/idx/unix.html`。）

35

Solaris 9 UNIX操作系统也是用以测试本书实例的操作系统之一。

2.3.7 其他UNIX系统

已经通过验证的其他UNIX系统的版本包括：

- AIX，IBM版的UNIX系统。
- HP-UX，HP版的UNIX系统。
- IRIX，Silicon Graphics版的UNIX系统。
- Unix Ware，SVR4派生的UNIX系统，现由SCO销售。

2.4 标准和实现的关系

前面提到的各个标准定义了任一实际系统的子集。本书主要关注4种实际的UNIX系统：FreeBSD 5.2.1、Linux 2.4.22、Mac OS X 10.3和Solaris 9。在这4种系统中，虽然只有Solaris 9能够称自己是一种UNIX系统，但是所有这4种系统都提供UNIX编程环境。因为所有这4种系统都在不同程度上依从POSIX，所以我们也将重点关注POSIX.1标准所要求的功能，并指出这4种系统的具体实现与POSIX之间的差别。仅仅一个特定实现所特有的功能和例程会被清楚地标记出来。因为SUS v3是POSIX.1的超集，所以我们还叙述了属于SUS v3但不属于POSIX.1的功能。

应当了解，这些实现都提供了对它们早期版本（例如SVR3.2和4.3BSD）功能的向后兼容性。例如，Solaris对POSIX规范中的非阻塞I/O（`O_NONBLOCK`）以及传统的系统V方法（`O_NDELAY`）都提供了支持。本书将只使用POSIX.1的功能，但是也会提及它所替代的是哪一种非标准功能。与此相类似，SVR3.2和4.3BSD以某种有别于POSIX.1标准的方法提供了可靠的信号机制。第10章将只说明POSIX.1的信号机制。

2.5 限制

UNIX系统实现定义了很多幻数和常量，其中有很多已被硬编码进程序中，或用特定的技术确定。由于大量标准化工作的努力，已有若干种可移植的方法用以确定这些幻数和实现定义的限制。这非常有助于软件的可移植性。

以下两种类型的限制是必需的：

(1) 编译时限制（例如，短整型的最大值是什么？）。

(2) 运行时限制（例如，文件名可以有多少个字符？）。

编译时限制可在头文件中定义，程序在编译时可以包含这些头文件。但是，运行时限制则要求进程调用一个函数以获得此种限制值。　36

另外，某些限制在一个给定的实现中可能是固定的(因此可以静态地在一个头文件中定义)，而在另一个实现上则可能是变化的（需要有一个运行时函数调用）。这类限制的一个例子是文件名的最大字符数。SVR4之前的系统V由于历史原因只允许文件名最多包含14个字符，而源于BSD的系统则将此增加到255。目前，大多数UNIX系统支持多文件系统类型，而每一种类型有它自己的限制。文件名的最大长度依赖于该文件处于何种文件系统中，例如，根文件系统中的文件名长度限制可能是14个字符，而在另一个文件系统中文件名长度限制可能是255个字符，这是运行时限制的一个例子。

为了解决这类问题，提供了以下三种限制：

(1) 编译时限制（头文件）。

(2) 不与文件或目录相关联的运行时限制（sysconf函数）。

(3) 与文件或目录相关联的运行时限制（pathconf和fpathconf函数）。

使事情变得更加复杂的是，如果一个特定的运行时限制在一个给定的系统上并不改变，则可将其静态地定义在一个头文件中。但是，如果没有将其定义在头文件中，则应用程序就必须调用三个conf函数中的一个（我们很快就会对它们进行说明），以确定其运行时的值。　37

2.5.1 ISO C限制

ISO C定义的限制都是编译时限制。表2-6列出了文件<limits.h>中定义的C标准限制。这些常量总是定义在头文件中，而且在一个给定系统中不会改变。第3列列出了ISO C标准可接受的最小值。这用于16位整型系统，使用1的补码表示。第4列列出了一个32位整型Linux系统的值，用2的补码表示。注意，对不带符号的数据类型都没有列出其最小值，它们都应为0。在64位系统中，其long整型的最大值与表中long long整型最大值相匹配。

我们将会遇到的一个不同之处是系统是否提供带符号或不带符号的字符值。从表2-6的第4列可以看出，该特定系统使用了带符号字符。从表中可以看到CHAR_MIN等于SCHAR_MIN，CHAR_MAX等于SCHAR_MAX。如果系统使用不带符号的字符，则CHAR_MIN等于0，CHAR_MAX等于UCHAR_MAX。

在头文件<float.h>中，对浮点数据类型也有类似的一组定义。如果读者在工作中涉及大量浮点数据类型，则应仔细查看该文件。

我们会遇到的另一个ISO C常量是FOPEN_MAX，这是具体实现保证可同时打开的标准I/O流的最小数。该值在头文件<stdio.h>中定义，其最小值是8。POSIX.1中的STREAM_MAX（若定义的话）必须具有与FOPEN_MAX相同的值。

表2-6 <limits.h>中定义的整型值大小

名 字	说 明	最小可接受值	典 型 值
CHAR_BIT	字符的位数	8	8
CHAR_MAX	字符的最大值	（见后）	127
CHAR_MIN	字符的最小值	（见后）	−128
SCHAR_MAX	带符号字符的最大值	127	127
SCHAR_MIN	带符号字符的最小值	−127	−128
UCHAR_MAX	不带符号字符的最大值	255	255
INT_MAX	整型的最大值	32 767	2 147 483 647
INT_MIN	整型的最小值	−32 767	−2 147 483 648
UINT_MAX	不带符号整型的最大值	65 535	4 294 967 295
SHRT_MIN	短整型的最小值	−32 767	−32 768
SHRT_MAX	短整型的最大值	32 767	32 767
USHRT_MAX	不带符号短整型的最大值	65 535	65 535
LONG_MAX	长整型的最大值	2 147 483 647	2 147 483 647
LONG_MIN	长整型的最小值	−2 147 483 647	−2 147 483 648
ULONG_MAX	不带符号长整型的最大值	4 294 967 295	4 294 967 295
LLONG_MAX	长长整型的最大值	9 223 372 036 854 775 807	9 223 372 036 854 775 807
LLONG_MIN	长长整型的最小值	−9 223 372 036 854 775 807	−9 223 372 036 854 775 808
ULLONG_MAX	不带符号长长整型的最大值	18 446 744 073 709 551 615	18 446 744 073 709 551 615
MB_LEN_MAX	多字节字符常量中的最大字节数	1	16

ISO C在<stdio.h>中还定义了常量TMP_MAX，这是由tmpnam函数产生的唯一文件名的最大数。关于此常量我们将在5.13节中进行更多说明。

在表2-7中，我们列出了本书所讨论的四种平台上的FOPEN_MAX和TMP_MAX值。

表2-7 在各种平台上的ISO限制

限 制	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
FOPEN_MAX	20	16	20	20
TMP_MAX	308 915 776	238 328	308 915 776	17 576

ISO C还定义了常量FILENAME_MAX，因为某些操作系统实现在历史上将它定义得太小，以至于不能满足应用的需求，所以我们应避免使用该常量。

2.5.2 POSIX限制

POSIX.1定义了很多涉及操作系统实现限制的常量，不幸的是，这是POSIX.1中最令人迷惑不解的部分之一。虽然POSIX.1定义了大量的限制和常量，我们只关心与基本POSIX.1接口有关的部分。这些限制和常量被分成下列5类。

38

(1) 不变的最小值：表2-8中的19个常量。

(2) 不变值：SSIZE_MAX。

(3) 运行时可以增加的值：CHARCLASS_NAME_MAX、COLL_WEIGHTS_MAX、 LINE_

MAX、NGROUPS_MAX以及RE_DUP_MAX。

(4) 运行时不变的值（可能不确定）：ARG_MAX、CHILD_MAX、HOST_NAME_MAX、LOGIN_NAME_MAX、OPEN_MAX、PAGESIZE、RE_DUP_MAX、STREAM_MAXS、SYMLOOP_MAX、TTY_NAME_MAX以及TZNAME_MAX。

(5) 路径名可变值（可能不确定）：FILESIZEBITS、LINK_MAX、MAX_CANON、MAX_INPUT、NAME_MAX、PATH_MAX、PIPE_BUF以及SYMLINK_MAX。

表2-8　<limits.h>中的POSIX.1不变最小值

名　字	说明：以下各项的最小可接受值	值
_POSIX_ARG_MAX	exec函数的参数长度	4 096
_POSIX_CHILD_MAX	每个实际用户ID的子进程数	25
_POSIX_HOST_NAME_MAX	gethostname函数返回的主机名最大长度	255
_POSIX_LINK_MAX	指向一个文件的链接数	8
_POSIX_LOGIN_NAME_MAX	登录名的最大长度	9
_POSIX_MAX_CANON	终端规范输入队列的字节数	255
_POSIX_MAX_INPUT	终端输入队列的可用空间	255
_POSIX_NAME_MAX	文件名中的字节数，不包括终止字符null	14
_POSIX_NGROUPS_MAX	每个进程同时的添加组 ID数	8
_POSIX_OPEN_MAX	每个进程的打开文件数	20
_POSIX_PATH_MAX	路径名中的字节数，包括终止字符null	256
_POSIX_PIPE_BUF	能原子地写到管道的字节数	512
_POSIX_RE_DUP_MAX	当使用间隔表示法\{m,n\}时，regexec和regcomp函数允许的基本正则表达式的重复出现次数	255
_POSIX_SSIZE_MAX	能存储在ssize_t对象中的值	32 767
_POSIX_STREAM_MAX	一个进程能同时打开的标准 I/O流数	8
_POSIX_SYMLINK_MAX	符号链接中的字节数	255
_POSIX_SYMLOOP_MAX	在解析路径名时，可遍历的符号链接数	8
_POSIX_TTY_NAME_MAX	终端设备名长度，包括终止字符null	9
_POSIX_TZNAME_MAX	时区名字节数	6

在这44个限制和常量中，有一些可定义在<limits.h>中，其余的则按具体条件可定义或不定义。在2.5.4节中说明sysconf、pathconf和fpathconf函数时，我们将描述可定义或不定义的限制和常量。在表2-8中，我们列出了19个不变最小值。

这些值是不变的——它们并不随系统而改变。它们指定了这些特征最具约束性的值。一个符合POSIX.1的实现应当提供至少这样大的值。这就是为什么将它们称为最小值的原因，虽然它们的名字都包含了MAX。另外，为了保证可移植性，一个严格遵循POSIX的应用程序不应要求更大的值。我们将在本书的适当部分说明每一个常量的含义。

39

一个严格遵循（strictly conforming）POSIX的应用程序有别于仅遵循POSIX（merely POSIX Confirming）的应用程序。一个遵循POSIX的应用程序只使用在IEEE标准1003.1-2001中定义的接口。一个严格遵循的应用程序也是遵循POSIX的，但除此之外它还应不依赖于POSIX未定义的行为，不使用任何已废弃的接口，以及不要求所使用的常量值大于表2-8中所列出的最小值。

不幸的是，这些不变最小值中的某一些在实际应用中太小了。例如，目前在大多数UNIX

系统中，每个进程可同时打开的文件数远远超过20。另外，_POSIX_PATH_MAX的最小限制值为255，这也太小了，路径名可能会超过这一限制。这意味着在编译时不能使用_POSIX_OPEN_MAX和_POSIX_PATH_MAX这两个常量作为数组长度。

表2-8中的每一个不变最小值都有一个相关的实现值，其名字是将表2-8中的名字删除前缀_POSIX_后构成的。不带有前导_POSIX_的名称打算作为给定实现支持的实际值（这19个实现值是本节开始部分所列出的2～5项：不变值、运行时可增加的值、运行时不变的值、以及路径名可变值）。问题是并不能确保所有这19个实现值都在<limits.h>头文件中定义。

例如，某个特定值可能不在此头文件中定义，其理由是：一个给定进程的实际值可能依赖于系统的存储总量。如果没有在头文件中定义这些值，则不能在编译时使用它们作为数组边界。所以，POSIX.1提供了三个运行时函数以供调用，它们是：sysconf、pathconf以及fpathconf。使用这三个函数可以在运行时得到实际的实现值。但是，还有一个问题，其中某些值由POSIX.1定义为“可能不确定的”（逻辑上无限的），这就意味着该值没有实际上限。例如在Linux中，readv或writev可用的iovec结构数仅受系统存储总量的限制。所以在Linux中，IOV_MAX被认为是不确定的。2.5.5节还将讨论运行时限制不确定的问题。

2.5.3 XSI限制

XSI还定义了处理实现限制的下面几个常量：

(1) 不变最小值：表2-9中列出的10个常量。

(2) 数值限制：LONG_BIT和WORD_BIT。

(3) 运行时不变值（可能不确定）：ATEXIT_MAX、IOV_MAX以及PAGE_SIZE。

不变最小值列示于表2-9中，在这些值中，很多与消息类有关。最后两个常量值例证了POSIX.1最小值太小的情况，根据推测这可能是考虑到了嵌入式POSIX.1实现。为此，Single UNIX Specification为遵循XSI的系统增加了具有较大最小值的符号。

40

表2-9 <limits.h>中的XSI不变最小值

名　字	说　明	最小可接受值	典 型 值
NL_ARGMAX	printf和scanf调用中的最大数字值	9	9
NL_LANGMAX	LANG环境变量中的最大字节数	14	14
NL_MSGMAX	最大消息数	32 767	32 767
NL_NMAX	*N*对1映射字符中的最大字节数	(未指定)	1
NL_SETMAX	最大集合数	255	255
NL_TEXTMAX	消息字符串中的最大字节数	_POSIX2_LINE_MAX	2 048
NZERO	默认的进程优先级	20	20
_XOPEN_IOV_MAX	readv或writev可使用的最大iovec结构数	16	16
_XOPEN_NAME_MAX	文件名中的字节数	255	255
_XOPEN_PATH_MAX	路径名中的字节数	1 024	1 024

2.5.4 sysconf、pathconf和fpathconf函数

我们已列出了实现必须支持的各种最小值，但是怎样才能找到一个特定系统实际支持的限制值呢？正如前面提到的，某些限制值在编译时是可用的，而另外一些则必须在运行时确定。我们也曾提及在一个给定的系统中某些限制值是不会更改的，而其他限制值则与文件和目录相

关联，是可以改变的。运行时限制可通过调用下面三个函数中的一个而取得。

```
#include <unistd.h>

long sysconf(int name);

long pathconf(const char *pathname, int name);

long fpathconf(int filedes, int name);
```

所有函数返回值：若成功则返回相应值；若出错则返回−1（见后）

后两个函数之间的差别是一个用路径名作为其参数，另一个则取文件描述符作为参数。

表2-10中列出了sysconf函数所使用的*name*参数，用于标识系统限制。以_SC_开始的常量用作标识运行时限制的sysconf参数。表2-11列出了pathconf和fpathconf函数为标识系统限制所使用的*name*参数。以_PC_开始的常量用作标识运行时限制的pathconf或fpathconf参数。

41

表2-10　sysconf的限制及*name*参数

限制名	说明	*name*参数
ARG_MAX	exec函数的参数最大长度（字节数）	_SC_ARG_MAX
ATEXIT_MAX	可用atexit函数登记的最大函数个数	_SC_ATEXIT_MAX
CHILD_MAX	每个实际用户ID的最大进程数	_SC_CHILD_MAX
clock ticks/second	每秒时钟滴答数	_SC_CLK_TCK
COLL_WEIGHTS_MAX	在本地定义文件中可以赋予LC_COLLATE顺序关键字项的最大权重数	_SC_COLL_WEIGHTS_MAX
HOST_NAME_MAX	gethostname函数返回的主机名最大长度	_SC_HOST_NAME_MAX
IOV_MAX	readv或writev函数可以使用的iovec结构的最大数	_SC_IOV_MAX
LINE_MAX	实用程序输入行的最大长度	_SC_LINE_MAX
LOGIN_NAME_MAX	登录名的最大长度	_SC_LOGIN_NAME_MAX
NGROUPS_MAX	每个进程同时添加的最大进程组ID数	_SC_NGROUPS_MAX
OPEN_MAX	每个进程的最大打开文件数	_SC_OPEN_MAX
PAGESIZE	系统存储页长度（字节数）	_SC_PAGESIZE
PAGE_SIZE	系统存储页长度（字节数）	_SC_PAGE_SIZE
RE_DUP_MAX	当使用间隔表示法\{m,n\}时，regexec和regcomp函数允许的基本正则表达式的重复出现次数	_SC_RE_DUP_MAX
STREAM_MAX	在任一时刻每个进程的最大标准I/O流数；如若定义，则其值一定与FOPEN_MAX相同	_SC_STREAM_MAX
SYMLOOP_MAX	在解析路径名期间，可遍历的符号链接数	_SC_SYMLOOP_MAX
TTY_NAME_MAX	终端设备名长度，包括终止字符null	_SC_TTY_NAME_MAX
TZNAME_MAX	时区名的最大字节数	_SC_TZNAME_MAX

我们需要更详细地说明这三个函数的不同返回值。

(1) 如果*name*不是表2-10和表2-11的第3列中的一个合适的常量，则所有这三个函数都会返回−1，并将errno设置为EINVAL。在本书后续部分都将使用这些限制常量。

(2) 有些*name*可以返回变量的值（返回值≥0），或者返回−1，这表示该值是不确定的，此

时并不改变errno的值。

(3) _SC_CLK_TCK的返回值是每秒钟的时钟滴答数，以用于times函数（见8.16节）的返回值。

42

表2-11　pathconf和fpathconf的限制及*name*参数

限　制　名	说　　明	*name*参数
FILESIZEBITS	以带符号整型值表示在指定目录中允许的普通文件最大长度所需的最少位数	_PC_FILESIZEBITS
LINK_MAX	文件链接数的最大值	_PC_LINK_MAX
MAX_CANON	终端规范输入队列的最大字节数	_PC_MAX_CANON
MAX_INPUT	终端输入队列可用空间的字节数	_PC_MAX_INPUT
NAME_MAX	文件名的最大字节数（不包括终止字符null）	_PC_NAME_MAX
PATH_MAX	相对路径名的最大字节数，包括终止字符null	_PC_PATH_MAX
PIPE_BUF	能原子地写到管道的最大字节数	_PC_PIPE_BUF
SYMLINK_MAX	符号链接中的字节数	_PC_SYMLINK_MAX

对于pathconf的参数*pathname*以及fpathconf的参数*filedes*有一些限制。如果不满足其中任何一个限制，则结果是未定义的。

(1) _PC_MAX_CANON和_PC_MAX_INPUT所引用的文件必须是终端文件。

(2) _PC_LINK_MAX所引用的文件可以是文件或目录。如果是目录，则返回值用于目录本身（而不是用于目录内的文件名项）。

(3) _PC_FILESIZEBITS和_PC_NAME_MAX所引用的文件必须是目录，返回值用于该目录中的文件名。

(4) _PC_PATH_MAX引用的文件必须是目录。当所指定的目录是工作目录时，返回值是相对路径名的最大长度（不幸的是，这不是我们想要知道的一个绝对路径名的实际最大长度，我们将在2.5.5节中再回到这一问题上来）。

(5) _PC_PIPE_BUF所引用的文件必须是管道、FIFO或目录。在管道或FIFO情况下，返回值是对所引用的管道或FIFO的限制值。对于目录，返回值是对在该目录中创建的任一FIFO的限制值。

(6) _PC_SYMLINK_MAX所引用的文件必须是目录。返回值是该目录中符号链接可能包含的字符串的最大长度。

43

实例

程序清单2-1中所示的awk(1)程序构建了一个C程序，它打印各pathconf和sysconf符号的值。

程序清单2-1　构建C程序以打印所有得到支持的系统配置限制

```
BEGIN   {
    printf("#include \"apue.h\"\n")
    printf("#include <errno.h>\n")
    printf("#include <limits.h>\n")
```

```
    printf("\n")
    printf("static void pr_sysconf(char *, int);\n")
    printf("static void pr_pathconf(char *, char *, int);\n")
    printf("\n")
    printf("int\n")
    printf("main(int argc, char *argv[])\n")
    printf("{\n")
    printf("\tif (argc != 2)\n")
    printf("\t\terr_quit(\"usage: a.out <dirname>\");\n\n")
    FS="\t+"
    while (getline <"sysconf.sym" > 0) {
        printf("#ifdef %s\n", $1)
        printf("\tprintf(\"%s defined to be %%d\\n\", %s+0);\n", $1, $1)
        printf("#else\n")
        printf("\tprintf(\"no symbol for %s\\n\");\n", $1)
        printf("#endif\n")
        printf("#ifdef %s\n", $2)
        printf("\tpr_sysconf(\"%s =\", %s);\n", $1, $2)
        printf("#else\n")
        printf("\tprintf(\"no symbol for %s\\n\");\n", $2)
        printf("#endif\n")
    }
    close("sysconf.sym")
    while (getline <"pathconf.sym" > 0) {
        printf("#ifdef %s\n", $1)
        printf("\tprintf(\"%s defined to be %%d\\n\", %s+0);\n", $1, $1)
        printf("#else\n")
        printf("\tprintf(\"no symbol for %s\\n\");\n", $1)
        printf("#endif\n")
        printf("#ifdef %s\n", $2)
        printf("\tpr_pathconf(\"%s =\", argv[1], %s);\n", $1, $2)
        printf("#else\n")
        printf("\tprintf(\"no symbol for %s\\n\");\n", $2)
        printf("#endif\n")
    }
    close("pathconf.sym")
    exit
}
END {
    printf("\texit(0);\n")
    printf("}\n\n")
    printf("static void\n")
    printf("pr_sysconf(char *mesg, int name)\n")
    printf("{\n")
    printf("\tlong  val;\n\n")
    printf("\tfputs(mesg, stdout);\n")
    printf("\terrno = 0;\n")
    printf("\tif ((val = sysconf(name)) < 0) {\n")
    printf("\t\tif (errno != 0) {\n")
    printf("\t\t\tif (errno == EINVAL)\n")
    printf("\t\t\t\tfputs(\" (not supported)\\n\", stdout);\n")
    printf("\t\t\telse\n")
    printf("\t\t\t\terr_sys(\"sysconf error\");\n")
    printf("\t\t} else {\n")
    printf("\t\t\tfputs(\" (no limit)\\n\", stdout);\n")
    printf("\t\t}\n")
    printf("\t} else {\n")
    printf("\t\tprintf(\" %%ld\\n\", val);\n")
    printf("\t}\n")
    printf("}\n\n")
```

44

```
    printf("static void\n")
    printf("pr_pathconf(char *mesg, char *path, int name)\n")
    printf("{\n")
    printf("\tlong  val;\n")
    printf("\n")
    printf("\tfputs(mesg, stdout);\n")
    printf("\terrno = 0;\n")
    printf("\tif ((val = pathconf(path, name)) < 0) {\n")
    printf("\t\tif (errno != 0) {\n")
    printf("\t\t\tif (errno == EINVAL)\n")
    printf("\t\t\t\tfputs(\" (not supported)\\n\", stdout);\n")
    printf("\t\t\telse\n")
    printf("\t\t\t\terr_sys(\"pathconf error, path = %%s\", path);\n")
    printf("\t\t} else {\n")
    printf("\t\t\tfputs(\" (no limit)\\n\", stdout);\n")
    printf("\t\t}\n")
    printf("\t} else {\n")
    printf("\t\tprintf(\" %%ld\\n\", val);\n")
    printf("\t}\n")
    printf("}\n")
}
```

该awk程序读取两个输入文件——pathconf.sym和sysconfig.sym，这两个文件中包含了用制表符分隔的限制名和符号列表。并非每种平台都会定义所有符号，所以围绕每个pathconf和sysconf调用，awk程序都使用了必要的#ifdef语句。

例如，awk程序将输入文件中类似于下列形式的行

```
NAME_MAX        _PC_NAME_MAX
```

45　转换成下列C代码：

```
#ifdef NAME_MAX
    printf("NAME_MAX is defined to be %d\n", NAME_MAX+0);
#else
    printf("no symbol for NAME_MAX\n");
#endif
#ifdef _PC_NAME_MAX
    pr_pathconf("NAME_MAX =", argv[1], _PC_NAME_MAX);
#else
    printf("no symbol for _PC_NAME_MAX\n");
#endif
```

程序清单2-2中的程序由awk产生，它会打印所有这些限制，并处理未定义限制的情况。

程序清单2-2　打印所有可能的sysconf和pathconf值

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

#ifdef ARG_MAX
    printf("ARG_MAX defined to be %d\n", ARG_MAX+0);
```

```
#else
    printf("no symbol for ARG_MAX\n");
#endif
#ifdef _SC_ARG_MAX
    pr_sysconf("ARG_MAX =", _SC_ARG_MAX);
#else
    printf("no symbol for _SC_ARG_MAX\n");
#endif

/* similar processing for all the rest of the sysconf symbols... */

#ifdef MAX_CANON
    printf("MAX_CANON defined to be %d\n", MAX_CANON+0);
#else
    printf("no symbol for MAX_CANON\n");
#endif
#ifdef _PC_MAX_CANON
    pr_pathconf("MAX_CANON =", argv[1], _PC_MAX_CANON);
#else
    printf("no symbol for _PC_MAX_CANON\n");
#endif

/* similar processing for all the rest of the pathconf symbols... */

    exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = sysconf(name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (not supported)\n", stdout);
            else
                err_sys("sysconf error");
        } else {
            fputs(" (no limit)\n", stdout);
        }
    } else {
        printf(" %ld\n", val);
    }
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = pathconf(path, name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (not supported)\n", stdout);
            else
                err_sys("pathconf error, path = %s", path);
        } else {
            fputs(" (no limit)\n", stdout);
        }
```

46

```
        } else {
            printf(" %ld\n", val);
        }
    }
```

表2-12总结了在本书讨论的四种系统上程序清单2-2的输出结果。“无符号”项表示该系统没有提供相应的_SC或_PC符号以查询相关常量值。因此，该限制是未定义的。与此相对照，“不受支持”项表示该符号由系统定义，但是未被sysconf或pathcon函数识别。“无限制”项表示该系统将相关常量定义为无限制，但并不表示该限制值可以是无限的。

47

表2-12 配置限制的实例

限 制	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	
				UFS文件系统	PCFS文件系统
ARG_MAX	65 536	131 072	262 144	1 048 320	1 048 320
ATEXIT_MAX	32	2 147 483 647	无符号	无限制	无限制
CHARCLASS_NAME_MAX	无符号	2 048	无符号	14	14
CHILD_MAX	867	999	100	7 877	7 877
clock ticks/second	128	100	100	100	100
COLL_WEIGHTS_MAX	0	255	2	10	10
FILESIZEBITS	不受支持	64	无符号	41	不受支持
HOST_NAME_MAX	255	不受支持	无符号	无符号	无符号
IOV_MAX	1 024	无限制	无符号	16	16
LINE_MAX	2 048	2 048	2 048	2 048	2 048
LINK_MAX	32 767	32 000	32 767	32 767	1
LOGIN_NAME_MAX	17	256	无符号	9	9
MAX_CANON	255	255	255	256	256
MAX_INPUT	255	255	255	512	512
NAME_MAX	255	255	765	255	8
NGROUPS_MAX	16	32	16	16	16
OPEN_MAX	1 735	1 024	256	256	256
PAGESIZE	4 096	4 096	4 096	8 192	8 192
PAGE_SIZE	4 096	4 096	无符号	8 192	8 192
PATH_MAX	1 024	4 096	1 024	1 024	1 024
PIPE_BUF	512	4 096	512	5 120	5 120
RE_DUP_MAX	255	32 767	255	255	255
STREAM_MAX	1 735	16	20	256	256
SYMLINK_MAX	不受支持	无限制	无符号	无符号	无符号
SYMLOOP_MAX	32	无限制	无符号	无符号	无符号
TTY_NAME_MAX	255	32	无符号	128	128
TZNAME_MAX	255	6	255	无限制	无限制

我们将在4.14节中看到，UFS是Berkeley快速文件系统的SVR4实现。PCFS是Solaris的MS-DOS FAT文件系统实现。 □

2.5.5 不确定的运行时限制

前面已提及某些限制值可能是不确定的。我们遇到的问题是：如果这些限制值没有在头文

件<limits.h>中定义，那么在编译时也就不能使用它们。但是，如果它们的值是不确定的，那么在运行时它们可能也是未定义的！让我们观察两种特殊的情况：为一个路径名分配存储区，以及确定文件描述符的数目。

1. 路径名

很多程序需要为路径名分配存储区。一般来说，在编译时就为其分配了存储区，而且不同的程序使用各种不同的幻数（其中很少是正确的）作为数组长度。例如，256、512、1024或标准I/O常量BUFSIZ。4.3BSD头文件<sys/param.h>中的常量MAXPATHLEN是正确值，但是很多4.3BSD应用程序并未使用它。 48

POSIX.1试图用PATH_MAX值来帮助我们，但是如果此值是不确定的，那么仍是毫无帮助的。程序清单2-3显示了一个全书中用来为路径名动态分配存储区的函数。

如若在<limits.h>中定义了常量PATH_MAX，那么就没有任何问题；如果没有定义，则需调用pathconf。因为pathconf的返回值是基于工作目录的相对路径名的最大长度，而工作目录是其第一个参数，所以，指定根目录为第一个参数，并将得到的返回值加1作为结果值。如果pathconf指明PATH_MAX是不确定的，那么我们就只能猜测某个值。

对于PATH_MAX是否在路径名末尾包括一个null字符这一点，SUS v3之前的标准表述得不清楚。出于安全方面的考虑，如果操作系统实现遵循先前的标准版本，则需要在为路径名分配的存储数量上加1。

处理不确定结果这种情况的正确方法与如何使用所分配的存储空间有关。例如，如果我们为getcwd调用分配空间（返回当前工作目录的绝对路径名，见4.22节），并且分配的空间太小，则返回一个出错，并将errno设置为ERANGE。然后可调用realloc来增加分配的空间（见7.8节和习题4.16）并重试。不断重复此操作，直到getcwd调用成功执行。

程序清单2-3　为路径名动态地分配空间

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

#ifdef  PATH_MAX
static int  pathmax = PATH_MAX;
#else
static int  pathmax = 0;
#endif

#define SUSV3   200112L

static long posix_version = 0;

/* If PATH_MAX is indeterminate, no guarantee this is adequate */
#define PATH_MAX_GUESS  1024

char *
path_alloc(int *sizep) /* also return allocated size, if nonnull */
{
    char    *ptr;
    int size;

    if (posix_version == 0)
        posix_version = sysconf(_SC_VERSION);

    if (pathmax == 0) {     /* first time through */
        errno = 0;
```

49

```
        if ((pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
                pathmax = PATH_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("pathconf error for _PC_PATH_MAX");
        } else {
            pathmax++;      /* add one since it's relative to root */
        }
    }
    if (posix_version < SUSV3)
        size = pathmax + 1;
    else
        size = pathmax;

    if ((ptr = malloc(size)) == NULL)
        err_sys("malloc error for pathname");

    if (sizep != NULL)
        *sizep = size;
    return(ptr);
}
```

2. 最大打开文件数

守护进程（daemon process，是指在后台运行且不与终端相连接的一种进程，也常被称为精灵进程或后台进程）中一个常见的代码序列是关闭所有打开的文件。某些程序中有下列形式的代码序列：

```
#include  <sys/param.h>

for (i = 0; i < NOFILE; i++)
    close(i);
```

这段程序假定在<sys/param.h>头文件中定义了常量NOFILE。另外一些程序则使用某些<stdio.h>版本提供作为上限的常量_NFILE。某些程序则将其上限值硬编码为20。

我们希望用POSIX.1的OPEN_MAX来确定此值以提高可移植性，但是，如果此值是不确定的，则仍然有问题。如果我们编写了下列代码：

```
#include  <unistd.h>

for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

而且如果OPEN_MAX是不确定的，那么for循环根本不会执行因为sysconf将返回−1。在这种情况下，最好的选择就是关闭所有描述符直至某个限制值（例如256）。如同上面的路径名实例一样，这并不能保证在所有情况下都能正确工作，但这却是我们所能选择的最好方法。程序清单2-4中使用了这种技术。

50

程序清单2-4　确定文件描述符数

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

#ifdef  OPEN_MAX
static long openmax = OPEN_MAX;
#else
static long openmax = 0;
```

```
#endif

/*
 * If OPEN_MAX is indeterminate, we're not
 * guaranteed that this is adequate.
 */
#define OPEN_MAX_GUESS  256

long
open_max(void)
{
    if (openmax == 0) {      /* first time through */
        errno = 0;
        if ((openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("sysconf error for _SC_OPEN_MAX");
        }
    }

    return(openmax);
}
```

我们可以耐心地调用close，直至得到一个出错返回，但是从close（EBADF）出错返回并不区分无效描述符和没有打开的描述符。如果试用此技术，而且描述符9未打开，描述符10却打开了，那么将停止在9上，而不会关闭10。dup函数（见3.12节）在超过了OPEN_MAX时会返回一个特定的出错值，但是用复制一个描述符一、二百次的方法来确定此值是一种非常极端的方法。

某些实现将返回LONG_MAX作为限制值，但这与不限制其值在效果上是相同的。Linux对ATEXIT_MAX所取的限制值就属于此种情况（见表2-12）。这将使程序的运行行为变得非常糟糕，因此它并不是一个好方法。

例如，我们可以使用内建在Bourne-again shell中的ulimit命令，来更改进程可同时打开的最大文件数。如果要将此限制值设置为在效果上是无限制的，那么通常要求具有特权（超级用户）。但是，一旦将其值设置为无穷大，sysconf就会将LONG_MAX报告为OPEN_MAX的限制值。程序若将此值作为要关闭的文件描述符数的上限（如程序清单2-4所示），那么为了试图关闭2 147 483 647个文件描述符，就会浪费大量的时间，实际上其中绝大多数文件描述符并未得到使用。

51

支持Single UNIX Specification中的XSI扩展的系统提供了getrlimit(2)函数（见7.11节）。它可用来返回一个进程可以同时打开的最大描述符数。使用该函数，我们能够检测出对于进程能够打开的文件数实际上并没有设置上限，于是也就避开了这个问题。

> OPEN_MAX被POSIX称为运行时不变值，这意味着在一个进程的生存期内其值不应发生变化。但是在支持XSI扩展的系统上，可以调用setrlimit(2)函数（见7.11节）更改一个运行进程的OPEN_MAX值（也可用C shell的limit命令或Bourne shell、Bourne-again shell和Korn shell的ulimit命令更改这个值）。如果系统支持这种功能，则可以更改程序清单2-4中的函数，使得每次调用此函数时都会调用sysconf，而不只是发生在第一次调用此函数时。

2.6 选项

表2-5中列出了POSIX.1的选项，并在2.2.3节中讨论了XSI选项组。如果我们要编写一些可移植的应用程序而这些程序与所有得到支持的选项有关，那么就需要一种可移植的方法以决定一种实现是否支持一个给定的选项。

如同对限制的处理（见2.5节）一样，Single UNIX Specification定义了三种处理方法：

(1) 编译时选项定义在<unistd.h>中。

(2) 与文件或目录无关的选项用sysconf函数确定。

(3) 与文件或目录有关的选项通过调用pathconf或fpathconf函数来发现。

选项包括列在表2-5的第3列中的符号，以及列在表2-13和表2-14中的符号。如若符号常量未定义，则必须使用sysconf、pathconf或fpathconf以决定该选项是否受到支持。在这种情况下，这些函数的*name*参数前缀_POSIX必须替换为_SC或_PC。对于以_XOPEN为前缀的常量，则在构成*name*参数时必须在其前放置_SC或_PC字符串。例如，若常量_POSIX_THREADS是未定义的，那么就可以将*name*参数设置为_SC_THREADS，并以此调用sysconf以确定该平台是否支持POSIX线程选项。如若常量_XOPEN_UNIX是未定义的，那么就可以通过将*name*参数设置为_SC_XOPEN_UNIX，来调用sysconf以确定该平台是否支持XSI扩展。

52

如果该平台定义了符号常量，则有以下三种可能：

(1) 如果符号常量的定义值为-1，那么该平台不支持相应的选项。

(2) 如果符号常量的定义值大于0，那么该平台支持相应的选项。

(3) 如果符号常量的定义值为0，则必须调用sysconf、pathconf或fpathconf以确定相应的选项是否受到支持。

除表2-5中已列出的那些选项之外，表2-13还总结了另外一些选项以及它们的符号常量，sysconf可以使用这些符号常量。

表2-13 sysconf的选项及*name*参数

选项名字	说明	*name* 参数
_POSIX_JOB_CONTROL	指明此实现是否支持作业控制	_SC_JOB_CONTROL
_POSIX_READER_WRITER_LOCKS	指明此实现是否支持读者-写者锁	_SC_READER_WRITER_LOCKS
_POSIX_SAVED_IDS	指明此实现是否支持saved set-user-ID和saved set-group-ID	_SC_SAVED_IDS
_POSIX_SHELL	指明此实现是否支持POSIX shell	_SC_SHELL
_POSIX_VERSION	指明POSIX.1版本	_SC_VERSION
_XOPEN_CRYPT	指明此实现是否支持XSI加密选项组	_SC_XOPEN_CRYPT
_XOPEN_LEGACY	指明此实现是否支持XSI遗留选项组	_SC_XOPEN_LEGACY
_XOPEN_REALTIME	指明此实现是否支持XSI实时选项组	_SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS	指明此实现是否支持XSI实时线程选项组	_SC_XOPEN_REALTIME_THREADS
_XOPEN_VERSION	指明XSI版本	_SC_XOPEN_VERSION

表2-14中总结了pathconf和fpathconf使用的符号常量。如同系统限制一样，关于

sysconf、pathconf和fpathconf如何处理选项，有如下几点值得注意：

(1) _SC_VERSION的返回值表示与该标准相关的年（以4位数表示）和月（以2位数表示）。该值可能是198808L、199009L、199506L，或者表示该标准后续版本的其他值，与SUS v3相关的值是200112L。

53

(2) _SC_XOPEN_VERSION的返回值表示该系统遵循的XSI版本。与SUS v3相关联的值是600。

(3) _SC_JOB_CONTROL、SC_SAVED_IDS以及_PC_VDISABLE的值不再表示可选功能。从SUS v3起，不再需要这些功能，但这些符号仍然被保留，以便向后兼容。

(4) 如果所指定的*pathname*或*filedes*不支持此功能，那么_PC_CHOWN_RESTRICTED和_PC_NO_TRUNC返回−1，而不会改变errno。

(5) _PC_CHOWN_RESTRICTED引用的文件必须是文件或者目录。如果是目录，那么返回值指明该选项是否可应用于该目录中的各个文件。

(6) _PC_NO_TRUNC引用的文件必须是一个目录。其返回值可用于该目录中的各文件名。

(7) _PC_VDISABLE引用的文件必须是一个终端文件。

表2-14　pathconf和fpathconf的选项及*name*参数

选项名字	说　明	*name* 参数
_POSIX_CHOWN_RESTRICTED	指明使用chown是否是受限的	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	指明路径名长于NAME_MAX是否会出错	_PC_NO_TRUNC
_POSIX_VDISABLE	若定义，可以用此值禁用终端特殊字符	_PC_VDISABLE
_POSIX_ASYNC_IO	指明对相关联的文件是否可以使用异步I/O	_PC_ASYNC_IO
_POSIX_PRIO_IO	指明对相关联的文件是否可以使用优先的I/O	_PC_PRIO_IO
_POSIX_SYNC_IO	指明对相关联的文件是否可以使用同步I/O	_PC_SYNC_IO

表2-15中列出了若干配置选项以及它们在本书所讨论的四个示例系统上的对应值。应该注意的是，有几个系统还没有跟上Single UNIX Specification的最新版本。例如，MAC OS X 10.3支持POSIX线程，但将_POSIX_THREADS定义为

```
#define _POSIX_THREADS
```

它没有指定一个值。为了遵循SUS v3，如若定义了该符号，那么其值应该设置为0、−1或200112。

表2-15　配置选项的实例

限　制	FreeBSD	Linux	Mac OS X	Solaris 9	
	5.2.1	2.4.22	10.3	UFS文件系统	PCFS文件系统
_POSIX_CHOWN_RESTRICTED	1	1	1	1	1
_POSIX_JOB_CONTROL	1	1	1	1	1
_POSIX_NO_TRUNC	1	1	1	1	不被支持
_POSIX_SAVED_IDS	不被支持	1	不被支持	1	1
_POSIX_THREADS	200112	200112	已定义	1	1
_POSIX_VDISABLE	255	0	255	0	0
_POSIX_VERSION	200112	200112	198808	199506	199506
_XOPEN_UNIX	不被支持	1	未定义	1	1
_XOPEN_VERSION	不被支持	500	未定义	3	3

如果未定义一个功能，也就是该系统未定义符号常量或对应的_SC或_PC名字，则将相关记录项标记为“未定义”。与此相对照，“已定义”记录项表示该符号常量已定义，但未指定值，前面的_POSIX_THREADS例子显示了这一点。如若系统定义了符号常量，但其值为-1或为0，但相应的sysconf或pathconf调用返回-1，那么该记录项将被标记为“不被支持”。

54

注意，当用于Solaris PCFS文件系统中的文件时，对于_PC_NO_TRUNC，pathconf返回值-1。PCFS系统支持DOS软盘格式，DOS文件名按DOS文件系统所要求的8.3格式截断，在进行此种操作时并无任何提示。

2.7 功能测试宏

如前所述，在头文件中定义了很多POSIX.1和XSI的符号。但是除了POSIX.1和XSI的定义之外，大多数实现在这些头文件中也加上了它们自己的定义。如果在编译一个程序时，希望它只使用POSIX的定义而不使用任何实现定义的限制，那么就需定义常量_POSIX_C_SOURCE。所有POSIX.1头文件中都使用此常量。当定义该常量时，就能排除任何实现专有的定义。

> POSIX.1标准的以前版本都定义了_POSIX_SOURCE常量。在POSIX.1的2001版中，它被替换为_POSIX_C_SOURCE。

常量_POSIX_C_SOURCE及_XOPEN_SOURCE被称为*功能测试宏*（feature test macro）。所有功能测试宏都以下划线开始。当要使用它们时，通常在cc命令行中以下列方式定义：

```
cc -D_POSIX_C_SOURCE=200112 file.c
```

这使得在C程序包括任何头文件之前，定义了功能测试宏。如果我们仅想使用POSIX.1定义，那么也可将源文件的第一行设置为：

```
#define _POSIX_C_SOURCE  200112
```

为使Single UNIX Specification v3的功能可由应用程序使用，需将常量_XOPEN_SOURCE定义为600。这与涉及POSIX.1功能时将_POSIX_C_SOURCE定义为200112L的作用相同。

55

Single UNIX Specification将c99实用程序定义为C编译环境的接口。随之，就可以用如下方式编译文件:

```
c99 -D_XOPEN_SOURCE=600 file.c -o file
```

为了在gcc C编译器中启用1999 ISO C扩展，可以使用-std = c99选项，如下所示：

```
gcc -D_XOPEN_SOURCE=600 -std=c99 file.c -o file
```

另一个功能测试宏是：_ _STDC_ _，它由符合ISO C标准的C编译器自动定义。这样就允许我们编写ISO C编译器和非ISO C编译器都能编译的程序。例如，为了利用ISO C原型功能（如果支持），一个头文件可能包含：

```
#ifdef __STDC__
void   *myfunc(const char *, int);
#else
void   *myfunc();
#endif
```

虽然，当今的大多数C编译器都支持ISO C标准，但在很多头文件中仍旧使用_ _STDC_ _功能测试宏。

2.8 基本系统数据类型

历史上，某些UNIX系统变量已与某些C数据类型联系在一起。例如，历史上主、次设备号一直存放在一个16位的短整型中，8位表示主设备号，另外8位表示次设备号。但是，很多较大的系统需要用多于256个值来表示其设备号，于是，就需要有一种不同的技术（实际上，Solaris用32位表示设备号：14位用于主设备号，18位用于次设备号）。

头文件<sys/types.h>中定义了某些与实现有关的数据类型，它们被称为*基本系统数据类型*（primitive system data type）。还有很多这种数据类型定义在其他头文件中。在头文件中，这些数据类型都是用C的typedef功能来定义的。它们绝大多数都以_t结尾。表2-16中列出了本书将使用的很多基本系统数据类型。

表2-16　某些常用的基本系统数据类型

类　型	说　　明
caddr_t	核心地址(14.9节)
clock_t	时钟滴答计数器（进程时间）（1.10节）
comp_t	压缩的时钟滴答（8.14节）
dev_t	设备号（主和次）（4.23节）
fd_set	文件描述符集（14.5.1节）
fpos_t	文件位置（5.10节）
gid_t	数值组ID
ino_t	i节点编号（4.14节）
mode_t	文件类型，文件创建模式（4.5节）
nlink_t	目录项的链接计数（4.14节）
off_t	文件大小和偏移量（带符号的）（lseek, 3.6节）
pid_t	进程ID和进程组ID（带符号的）（8.2节和9.4节）
ptrdiff_t	两个指针相减的结果（带符号的）
rlim_t	资源限制（7.11节）
sig_atomic_t	能原子地访问的数据类型（10.15节）
sigset_t	信号集（10.11节）
size_t	对象（例如字符串）大小（不带符号的）（3.7节）
ssize_t	返回字节计数的函数（带符号的）（read、write, 3.7节）
time_t	日历时间的秒计数器（1.10节）
uid_t	数值用户ID
wchar_t	能表示所有不同的字符码

用这种方式定义了这些数据类型后，就不再需要考虑因系统而异的程序实现细节。在本书中涉及这些数据类型时，我们会说明为什么要使用它们。

2.9 标准之间的冲突

就整体而言，这些不同的标准之间配合得相当好。但是我们也很关注它们之间的差别，特别是ISO C标准和POSIX.1之间的差别，而它们之间也确实有一些差别（因为SUS v3是POSIX.1的超集，所以不对它进行特别的说明）。

56

ISO C定义了函数clock，它返回进程使用的CPU时间，返回值类型是clock_t。为了将

此值变换成以秒为单位，将其除以在<time.h>头文件中定义的CLOCKS_PER_SEC。POSIX.1定义了函数times，它返回其调用者及其所有终止子进程的CPU时间以及时钟时间，所有这些值都是clock_t类型值。sysconf函数用来获取每秒钟的滴答数，用于表示times函数的返回值。有一个相同的术语，即每秒钟的滴答数，但ISO C和POSIX.1的定义却不同。这两个标准也用同一数据类型（clock_t）来保存这些不同的值，这种差别可以在Solaris中看到，其中clock返回微秒数（因此，CLOCK_PER_SEC是一百万），而sysconf为每秒钟的滴答数返回的值是100。

另一个可能产生冲突的领域是：在ISO C标准定义函数时，可能没有考虑到POSIX.1的某些要求。在POSIX环境下，有些函数可能要求有一个与C环境下不同的实现，因为POSIX环境中有多个进程，而ISO C环境则很少考虑主机操作系统。尽管如此，很多依从POSIX的系统为了兼容性也会实现ISO C函数。signal函数就是一个例子。如果在不了解的情况下使用了Solaris所提供的signal函数（希望编写可在ISO C环境和较早的UNIX系统中运行的可移植代码），那么它将提供与POSIX.1 sigaction函数不同的语义。第10章将对signal函数作更多说明。 [57]

2.10 小结

在过去20年中，在UNIX编程环境的标准化方面已经取得了很大进展。本章对ISO C、POSIX和Single UNIX Specification三个主要标准进行了说明，也分析了这些标准对本书主要关注的四个实现即FreeBSD、Linux、Mac OS X和Solaris所产生的影响。这些标准都试图定义一些可能随实现而更改的参数，但是我们已经看到这些限制并不完善。本书将涉及很多这些限制和幻常量。

在本书最后的参考书目中，说明了如何获得这些标准的副本的方法。

习题

2.1 在2.8节中提到，一些基本系统数据类型可以在多个头文件中定义。例如，在FreeBSD 5.2.1中，size_t在26个不同的头文件中都有定义。由于一个程序可能包含这26个不同的头文件，并且ISO C不允许对同一个名字进行多次类型定义，因此该如何编写这些头文件？

2.2 检查系统的头文件，列出实现基本系统数据类型所用到的实际数据类型。

2.3 改写程序清单2-4中的程序，使其在sysconf为OPEN_MAX限制返回LONG_MAX时，避免进行不必要的处理。 [58]

第3章

文 件 I/O

3.1 引言

本章开始讨论UNIX系统，先说明可用的文件I/O函数——打开文件、读文件、写文件等。UNIX系统中的大多数文件I/O只需用到5个函数：open、read、write、lseek以及close。然后说明不同缓冲区长度对read和write函数的影响。

本章所说明的函数经常被称为不带缓冲的I/O（unbuffered I/O，与将在第5章中说明的标准I/O例程相对照）。术语不带缓冲指的是每个read和write都调用内核中的一个系统调用。这些不带缓冲的I/O函数不是ISO C的组成部分，但是，它们是POSIX.1和Single UNIX Specification的组成部分。

只要涉及在多个进程间共享资源，原子操作的概念就变得非常重要。我们将通过文件I/O和open函数的参数来讨论此概念。然后，本章将进一步讨论在多个进程间如何共享文件，以及所涉及的内核数据结构。在讨论了这些特征后，将说明dup、fcntl、sync、fsync和ioctl函数。

3.2 文件描述符

对于内核而言，所有打开的文件都通过文件描述符引用。文件描述符是一个非负整数。当打开一个现有文件或创建一个新文件时，内核向进程返回一个文件描述符。当读或写一个文件时，使用open或creat返回的文件描述符标识该文件，将其作为参数传送给read或write。

按照惯例，UNIX系统shell使用文件描述符0与进程的标准输入相关联，文件描述符1与标准输出相关联，文件描述符2与标准出错输出相关联。这是各种 shell以及很多应用程序使用的惯例，而与UNIX内核无关。尽管如此，如果不遵照这种惯例，那么很多UNIX系统应用程序就不能正常工作。

在依从POSIX的应用程序中，幻数0、1、2应当替换成符号常量STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO。这些常量都定义在头文件<unistd.h>中。

文件描述符的变化范围是0～OPEN_MAX（见表2-10）。早期的UNIX系统实现采用的上限值是19（允许每个进程最多打开20个文件），但现在很多系统则将其增至63。

> 对于FreeBSD 5.2.1、Mac OS X 10.3以及Solaris 9，文件描述符的变化范围实际上是无限的，它只受到系统配置的存储器总量、整型的字长以及系统管理员所配置的软限制和硬限制的约束。Linux 2.4.22对于每个进程的文件描述符数的硬限制是1 048 576。

3.3 open函数

调用open函数可以打开或创建一个文件。

```
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /* mode_t mode */ );
```

返回值：若成功则返回文件描述符，若出错则返回−1

我们将第三个参数写为...，ISO C用这种方法表明余下参数的数量及其类型根据具体的调用会有所不同。对于open函数而言，仅当创建新文件时才使用第三个参数（稍后将对此进行说明）。在函数原型中将此参数放置在注释中。

*pathname*是要打开或创建文件的名字。*oflag*参数可用来说明此函数的多个选项。用下列一个或多个常量进行"或"运算构成*oflag*参数（这些常量定义在<fcntl.h>头文件中）：

O_RDONLY　　只读打开。

O_WRONLY　　只写打开。

O_RDWR　　读、写打开。

> 大多数实现将O_RDONLY定义为0，O_WRONLY定义为1，O_RDWR定义为2，以与早期的程序兼容。

在这三个常量中必须指定一个且只能指定一个。下列常量则是可选择的：

60

O_APPEND　　每次写时都追加到文件的尾端。3.11节将详细说明此选项。

O_CREAT　　若此文件不存在，则创建它。使用此选项时，需要第三个参数*mode*，用其指定该新文件的访问权限位（4.5节将说明文件的访问权限位，那时就能了解如何指定*mode*，以及如何用进程的umask值修改它）。

O_EXCL　　如果同时指定了O_CREAT，而文件已经存在，则会出错。用此可以测试一个文件是否存在，如果不存在，则创建此文件，这使测试和创建两者成为一个原子操作。3.11节将更详细地说明原子操作。

O_TRUNC　　如果此文件存在，而且为只写或读写成功打开，则将其长度截短为0。

O_NOCTTY　　如果*pathname*指的是终端设备，则不将该设备分配作为此进程的控制终端。9.6节将说明控制终端。

O_NONBLOCK　　如果*pathname*指的是一个FIFO、一个块特殊文件或一个字符特殊文件，则此选项为文件的本次打开操作和后续的I/O操作设置非阻塞模式。14.2节将说明此工作模式。

> 较早的系统V版本引入了O_NDELAY（不延迟）标志，它与O_NONBLOCK（不阻塞）选项类似，但它的读操作返回值具有二义性。如果不能从管道、FIFO或设备读取数据，则不延迟选项使read返回0，这与表示已读到文件尾端的返回值0相冲突。基于SVR4的系统仍支持这种语义的不延迟选项，但是新的应用程序应当使用不阻塞选项代替之。

下面三个标志也是可选的。它们是Single UNIX Specification（以及POSIX.1）中同步输入和输出选项的一部分。

O_DSYNC　　使每次write等待物理I/O操作完成，但是如果写操作并不影响读取刚写入的数据，则不等待文件属性被更新。

O_RSYNC　使每一个以文件描述符作为参数的read操作等待，直至任何对文件同一部分进行的未决写操作都完成。

O_SYNC　使每次write都等到物理I/O操作完成，包括由write操作引起的文件属性更新所需的I/O。3.14节将使用此选项。

O_DSYNC和O_SYNC标志有微妙的区别。仅当文件属性需要更新以反映文件数据变化（例如，更新文件大小以反映文件中包含了更多的数据）时，O_DSYNC标志才影响文件属性。而设置O_SYNC标志后，数据和属性总是同步更新。当文件用O_DSYN标志打开，在重写其现有的部分内容时，文件时间属性不会同步更新。与此相反，如果文件是用O_SYNC标志打开，那么对该文件的每一次write操作都将在write返回前更新文件时间，这与是否改写现有字节或增写文件无关。

Solaris 9支持所有这三个标志。FreeBSD 5.2.1和 Mac OS X 10.3设置了另外一个标志（O_FSYNC），它与标志O_SYNC的作用相同。因为这两个标志是等效的，所以FreeBSD 5.2.1定义它们为同一值（但奇怪的是，Mac OS X 10.3没有定义O_SYNC）。FreeBSD 5.2.1和Mac OS X 10.3不支持O_DSYNC或O_RSYNC标志。Linux 2.4.22将它们处理成与O_SYNC相同。

61

由open返回的文件描述符一定是最小的未用描述符数值。这一点被某些应用程序用来在标准输入、标准输出或标准出错输出上打开新的文件。例如，一个应用程序可以先关闭标准输出（通常是文件描述符1），然后打开另一个文件，执行打开操作前就能了解到该文件一定会在文件描述符1上打开。在3.12节说明dup2函数时，可以了解到有更好的方法来保证在一个给定的描述符上打开一个文件。

文件名和路径名截短

如果NAME_MAX是14，而我们却试图在当前目录中创建一个其文件名包含15个字符的新文件，此时会发生什么呢？按照传统，早期的系统V版本（例如SVR2）允许这种使用方法，但总是将文件名截短为14个字符，而且不给出任何信息，而BSD类的系统则返回出错状态，并将errno设置为ENAMETOOLONG。无声无息地截短文件名会引起问题，而且它不仅仅影响到创建新文件。如果NAME_MAX是14，并且存在一个其文件名恰好就是14个字符的文件，那么以*pathname*作为其参数的任一函数（open、stat等）都无法确定该文件的原始名是什么？其原因是这些函数无法判断该文件名是否被截短过。

在POSIX.1中，常量_POSIX_NO_TRUNC决定了是要截短过长的文件名或路径名，还是返回一个出错。正如我们在第2章中已见过的，根据文件系统的类型，此值可以变化。

是否返回一个出错值在很大程度上是历史形成的。例如，基于SVR4的系统对传统的系统V文件系统（S5）并不产生出错返回，但是它对BSD风格的文件系统（UFS）则产生出错返回。

作为另一个例子，参见表2-15。Solaris对UFS返回出错，对与DOS兼容的文件系统PCFS则不返回出错，其原因是DOS会无声无息地截短不适配8.3格式的文件名。BSD类系统和Linux总是会返回出错。

若_POSIX_NO_TRUNC有效，则在整个路径名超过PATH_MAX，或路径名中的任一文件名超过NAME_MAX时，返回出错状态，并将errno设置为ENAMETOOLONG。

3.4 creat函数

也可调用creat函数创建一个新文件。

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

返回值：若成功则返回为只写打开的文件描述符，若出错则返回−1

62

注意，此函数等效于：

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

> 在早期的UNIX系统版本中，open的第二个参数只能是0、1或2。没有办法打开一个尚未存在的文件，因此需要另一个系统调用creat以创建新文件。现在，open函数提供了选项O_CREAT和O_TRUNC，于是也就不再需要creat函数。

在4.5节中，我们将详细说明文件访问权限（file's access permission），并说明如何指定*mode*。

creat的一个不足之处是它以只写方式打开所创建的文件。在提供open的新版本之前，如果要创建一个临时文件，并要先写该文件，然后又读该文件，则必须先调用creat、close，然后再调用open。现在则可用下列方式调用open：

```
open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5 close函数

可调用close函数关闭一个打开的文件：

```
#include <unistd.h>

int close(int filedes);
```

返回值：若成功则返回0，若出错则返回−1

关闭一个文件时还会释放该进程加在该文件上的所有记录锁。14.3节将讨论这一点。

当一个进程终止时，内核自动关闭它所有打开的文件。很多程序都利用了这一功能而不显式地用close关闭打开文件。实例见程序清单1-2中的程序。

3.6 lseek函数

每个打开的文件都有一个与其相关联的“当前文件偏移量”（current file offset）。它通常是一个非负整数，用以度量从文件开始处计算的字节数（本节稍后将对“非负”这一修饰词的某些例外进行说明）。通常，读、写操作都从当前文件偏移量处开始，并使偏移量增加所读写的字节数。按系统默认的情况，当打开一个文件时，除非指定O_APPEND选项，否则该偏移量被设置为0。

可以调用lseek显式地为一个打开的文件设置其偏移量。

```
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
```

返回值：若成功则返回新的文件偏移量，若出错则返回−1

63

对参数*offset*的解释与参数*whence*的值有关。

- 若*whence*是SEEK_SET，则将该文件的偏移量设置为距文件开始处*offset*个字节。

• 若*whence*是SEEK_CUR，则将该文件的偏移量设置为其当前值加*offset*，*offset*可为正或负。
• 若*whence*是SEEK_END，则将该文件的偏移量设置为文件长度加*offset*，*offset*可为正或负。

若lseek成功执行，则返回新的文件偏移量，为此可以用下列方式确定打开文件的当前偏移量：

```
off_t    currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

这种方法也可用来确定所涉及的文件是否可以设置偏移量。如果文件描述符引用的是一个管道、FIFO或网络套接字，则lseek返回-1，并将errno设置为ESPIPE。

> 三个符号常量SEEK_SET、SEEK_CUR和SEEK_END是由系统V引入的。在系统V之前，*whence*被指定为0（绝对偏移量）、1（相对于当前位置的偏移量）或2（相对文件尾端的偏移量）。现有的很多软件仍然把这些数字直接写在代码里。
>
> lseek中的字符l表示长整型。在引入off_t数据类型之前，*offset*参数和返回值是长整型。lseek是由V7引入的，当时C语言中增加了长整型（在V6中，用函数seek和tell提供类似功能）。

实 例

程序清单3-1中的程序用于测试能否对其标准输入设置偏移量。

程序清单3-1 测试能否对标准输入设置偏移量

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

如果用交互方式调用此程序，则可得：

```
$ ./a.out < /etc/motd
seek OK
$ cat < /etc/motd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```

64

通常，文件的当前偏移量应当是一个非负整数，但是，某些设备也可能允许负的偏移量。但对于普通文件，则其偏移量必须是非负值。因为偏移量可能是负值，所以在比较lseek的返回值时应当谨慎，不要测试它是否小于0，而要测试它是否等于-1。

> 在Intel x86处理器上运行的FreeBSD上的/dev/kmem设备支持负的偏移量。
>
> 因为偏移量（off_t）是带符号数据类型（见表2-16），所以文件最大长度会减少一半。例如，若off_t是32位整型，则文件最大长度是$2^{31}-1$个字节。

lseek仅将当前的文件偏移量记录在内核中，它并不引起任何I/O操作。然后，该偏移量用

于下一个读或写操作。

文件偏移量可以大于文件的当前长度，在这种情况下，对该文件的下一次写将加长该文件，并在文件中构成一个空洞，这一点是允许的。位于文件中但没有写过的字节都被读为0。

文件中的空洞并不要求在磁盘上占用存储区。具体处理方式与文件系统的实现有关，当定位到超出文件尾端之后写时，对于新写的数据需要分配磁盘块，但是对于原文件尾端和新开始写位置之间的部分则不需要分配磁盘块。

程序清单3-2中的程序用于创建一个具有空洞的文件。

程序清单3-2　创建一个具有空洞的文件

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int     fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

65

运行该程序得到：

```
$ ./a.out
$ ls -l file.hole                    检查其大小
-rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
$ od -c file.hole                    观察实际内容
0000000   a  b  c  d  e  f  g  h  i  j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000   A  B  C  D  E  F  G  H  I  J
0040012
```

使用od(1)命令观察该文件的实际内容。命令行中的-c标志表示以字符方式打印文件内容。从中可以看到，文件中间的30个未写字节都被读成为0。每一行开始的一个7位数是以八进制形式表示的字节偏移量。

为了证明在该文件中确实有一个空洞，将刚创建的文件与具有同样长度但无空洞的文件进行比较：

```
$ ls -ls file.hole file.nohole     比较长度
  8 -rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
 20 -rw-r--r--  1 sar       16394 Nov 25 01:03 file.nohole
```

虽然两个文件的长度相同，但无空洞的文件占用了20个磁盘块，而具有空洞的文件只占用8个磁盘块。

在此实例中调用了将在3.8节中说明的write函数。4.12节将对具有空洞的文件进行更多说明。□

因为lseek使用的偏移量是用off_t类型表示的，所以允许具体实现根据各自特定的平台自行选择大小合适的数据类型。现今大多数平台提供两组接口以处理文件偏移量：一组使用32位文件偏移量，另一组则使用64位文件偏移量。

Single UNIX Specification向应用程序提供了一种方法，使其通过sysconf函数确定何种环境受到支持（见2.5.4节）。表3-1总结了定义的sysconf常量。

表3-1　sysconf的数据大小选项和*name*参数

选 项 名 字	说　明	*name*参数
_POSIX_V6_ILP32_OFF32	int、long、pointer和off_t类型是32位	_SC_V6_ILP32_OFF32
_POSIX_V6_ILP32_OFFBIG	int、long和pointer类型是32位；off_t类型至少是64位	_SC_V6_ILP32_OFFBIG
_POSIX_V6_LP64_OFF64	int类型是32位；long、pointer和off_t类型是64位	_SC_V6_LP64_OFF64
_POSIX_V6_LP64_OFFBIG	int类型是32位；long、pointer和off_t类型至少是64位	_SC_V6_LP64_OFFBIG

66

c99编译器要求使用getconf(1)命令，将所希望的数据大小模型映射为编译和连接程序所需的标志。根据每个平台支持环境的不同，可能需要不同的标志和库。

不幸的是，在这一方面，实现没有跟上标准的步伐。使人更迷惑不解的是Single UNIX Specification第2版和第3版之间更改了若干个名字。

为了避开这一点，应用程序可将符号常量_FILE_OFFSET_BITS设置为64，以支持64位偏移量。这样处理后就将off_t定义更改为64位带符号整型。将_FILE_OFFSET_BITS符号常量设置为32，就支持32位文件偏移量。但是，应当知道的是：虽然本书讨论的四种平台都支持32位和64位文件偏移量，其方法是将_FILE_OFFSET_BITS符号常量设置为所希望的值，但并不保证这是可移植的。

注意：尽管可以支持64位文件偏移量，但是否能创建一个大于2 TB（$2^{31}-1$个字节）的文件则依赖于底层文件系统的类型。

3.7　read函数

调用read函数从打开文件中读数据。

```
#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbytes);
```

返回值：若成功则返回读到的字节数，若已到文件结尾则返回0，若出错则返回-1

如read成功，则返回读到的字节数。如已到达文件结尾，则返回0。

有多种情况可使实际读到的字节数少于要求读的字节数：

- 读普通文件时，在读到要求字节数之前已到达了文件尾端。例如，若在到达文件尾端之前还有30个字节，而要求读100个字节，则read返回30，下一次再调用read时，它将返

回0（文件尾端）。

- 当从终端设备读时，通常一次最多读一行（第18章将介绍如何改变这一点）。
- 当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。
- 当从管道或FIFO读时，如若管道包含的字节少于所需的数量，那么read将只返回实际可用的字节数。
- 当从某些面向记录的设备（例如磁带）读时，一次最多返回一个记录。

67

- 当某一信号造成中断，而已经读了部分数据量时。我们将在10.5节进一步讨论此种情况。

读操作从文件的当前偏移量处开始，在成功返回之前，该偏移量将增加实际读到的字节数。

POSIX.1从几个方面对read函数的原型作了更改。其经典定义是：

```
int read(int filedes, char *buf, unsigned nbytes);
```

- 首先，为了与ISO C保持一致，将第二个参数由char *改为void *。在ISO C中，类型void *用于表示通用指针。
- 其次，其返回值必须是一个带符号整数（ssize_t），以返回正字节数、0（表示文件尾端）或-1（出错）。
- 最后，第三个参数在历史上是一个不带符号整数，这允许一个16位的实现一次读或写的数据可以多达65 534个字节。在1990 POSIX.1标准中，引入了基本系统数据类型ssize_t以提供带符号的返回值，不带符号的size_t则用于第三个参数（见2.5.2节中的SSIZE_MAX常量）。

3.8 write函数

调用write函数向打开的文件写数据。

```
#include <unistd.h>

ssize_t write(int filedes, const void *buf, size_t nbytes);

                        返回值：若成功则返回已写的字节数，若出错则返回-1
```

其返回值通常与参数*nbytes*的值相同，否则表示出错。write出错的一个常见原因是：磁盘已写满，或者超过了一个给定进程的文件长度限制（见7.11节及习题10.11）。

对于普通文件，写操作从文件的当前偏移量处开始。如果在打开该文件时，指定了O_APPEND选项，则在每次写操作之前，将文件偏移量设置在文件的当前结尾处。在一次成功写之后，该文件偏移量增加实际写的字节数。

3.9 I/O的效率

程序清单3-3中的程序使用read和write函数复制一个文件。关于该程序应注意下列各点：

- 它从标准输入读，写至标准输出，这就假定在执行本程序之前，这些标准输入、输出已由shell安排好。确实，所有常用的UNIX系统shell都提供一种方法，它在标准输入上打开一个文件用于读，在标准输出上创建（或重写）一个文件。这使得程序不必自行打开输入和输出文件。

68

- 很多应用程序假定标准输入是文件描述符0，标准输出是文件描述符1。本示例中则使用在<unistd.h>中定义的两个名字：STDIN_FILENO和STDOUT_FILENO。

- 考虑到进程终止时，UNIX系统内核会关闭该进程的所有打开的文件描述符，所以此示例并不会关闭输入和输出文件。
- 对UNIX系统内核而言，文本文件和二进制代码文件并无区别，所以本示例对这两种文件都能工作。

程序清单3-3 将标准输入复制到标准输出

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

我们没有回答的一个问题是如何选取BUFFSIZE值。在回答此问题之前，让我们先用各种不同的BUFFSIZE值来运行此程序。表3-2显示了用20种不同的缓冲区长度，读103 316 352字节的文件所得到的结果。

表3-2 用不同缓冲区长度进行读操作的计时结果

BUFFSIZE	用户CPU（秒）	系统CPU（秒）	时钟时间（秒）	循环次数
1	124.89	161.65	288.64	103 316 352
2	63.10	80.96	145.81	51 658 176
4	31.84	40.00	72.75	25 829 088
8	15.17	21.01	36.85	12 914 544
16	7.86	10.27	18.76	6 457 272
32	4.13	5.01	9.76	3 228 636
64	2.11	2.48	6.76	1 614 318
128	1.01	1.27	6.82	807 159
256	0.56	0.62	6.80	403 579
512	0.27	0.41	7.03	201 789
1 024	0.17	0.23	7.84	100 894
2 048	0.05	0.19	6.82	50 447
4 096	0.03	0.16	6.86	25 223
8 192	0.01	0.18	6.67	12 611
16 384	0.02	0.18	6.87	6 305
32 768	0.00	0.16	6.70	3 152
65 536	0.02	0.19	6.92	1 576
131 072	0.00	0.16	6.84	788
262 144	0.01	0.25	7.30	394
524 288	0.00	0.22	7.35	198

用程序清单3-3中的程序读文件，其标准输出被重新定向到/dev/null上。此测试所用的文件系统是Linux ext2文件系统，其块长为4 096字节（块长由st_blksize表示，在4.12节中说明其值为4 096）。系统CPU时间的最小值出现在BUFFSIZE为4096处，继续增加缓冲区长度对此时间几乎没有影响。

大多数文件系统为改善其性能都采用某种预读（read ahead）技术。当检测到正进行顺序读取时，系统就试图读入比应用程序所要求的更多数据，并假想应用程序很快就会读这些数据。从表3-2中最后几个记录项可以观察到，在ext2中，当BUFFSIZE为128 KB后，预读停止了，这对读操作的性能产生了影响。

我们以后还将回到这一实例上。3.14节将用此说明同步写的效果，5.8节将比较不带缓冲的I/O时间与标准I/O库所用的时间。

69

应当了解，在什么时间对实施文件读、写操作的程序进行性能度量。操作系统试图用缓存技术将相关文件放置在主存中，所以如若重复度量程序性能，那么后续运行该程序所得到的计时很可能好于第一次。其原因是，第一次运行使文件进入系统缓存，后续各次运行一般从系统缓存访问文件，而无需读、写磁盘。

在表3-2所示的测试数据中，每次运行使用不同的缓冲区大小和不同的文件副本，所以前一次运行在缓存中留下的数据是后一次运行所不需要的，换言之，后一次运行不会在缓存中找到它所需要的数据。这些文件都足够大，不可能全部保留在缓存中（测试系统配置了512 MB RAM）。

3.10 文件共享

UNIX系统支持在不同进程间共享打开的文件。在介绍dup函数之前，先要说明这种共享。为此先介绍内核用于所有I/O的数据结构。

下面的说明是概念性的，与特定实现可能匹配，也可能不匹配。参阅Bach[1986]对系统V中相关数据结构的讨论。McKusick等人[1996]说明了4.4BSD中的相关数据结构。McKusick和Neville-Neil[2005]对FreeBSD 5.2进行了介绍。对Solaris的类似讨论参见Marno和McDougall[2001]。

70

内核使用三种数据结构表示打开的文件，它们之间的关系决定了在文件共享方面一个进程对另一个进程可能产生的影响。

(1) 每个进程在进程表中都有一个记录项，记录项中包含有一张打开文件描述符表，可将其视为一个矢量，每个描述符占用一项。与每个文件描述符相关联的是：

- (a) 文件描述符标志（close_on_exec，参见图3-1和3.14节）。
- (b) 指向一个文件表项的指针。

(2) 内核为所有打开文件维持一张文件表。每个文件表项包含：

- (a) 文件状态标志（读、写、添写、同步和非阻塞等，关于这些标志的更多信息参见3.14节）。
- (b) 当前文件偏移量。
- (c) 指向该文件v节点表项的指针。

(3) 每个打开文件（或设备）都有一个v节点（v-node）结构。v节点包含了文件类型和对此文件进行各种操作的函数的指针。对于大多数文件，v节点还包含了该文件的i节点（i-node，索引节点）。这些信息是在打开文件时从磁盘上读入内存的，所以所有关于文件的信息都是快

速可供使用的。例如，i节点包含了文件的所有者、文件长度、文件所在的设备、指向文件实际数据块在磁盘上所在位置的指针等等（4.14节较详细地说明了典型UNIX系统文件系统，并将更多地介绍i节点）。

> Linux没有使用v节点，而是使用了通用i节点结构。虽然两种实现有所不同，但在概念上，v节点与i节点是一样的。两者都指向文件系统特有的i节点结构。

我们忽略了某些实现细节，但这并不影响我们的讨论。例如，打开文件描述符表可存放在用户空间，而非进程表中。这些表也可以用多种方式实现，不必一定是数组；例如，可将它们实现为结构的链接表。这些细节并不影响我们在文件共享方面的讨论。

图3-1显示了一个进程的三张表之间的关系。该进程有两个不同的打开文件：一个文件打开为标准输入（文件描述符0），另一个打开为标准输出（文件描述符为1）。从UNIX系统的早期版本[Thompson 1978]以来，这三张表之间的基本关系一直保持至今。这种安排对于在不同进程之间共享文件的方式非常重要。在以后的章节中涉及其他文件共享方式时还会回到这张图上来。

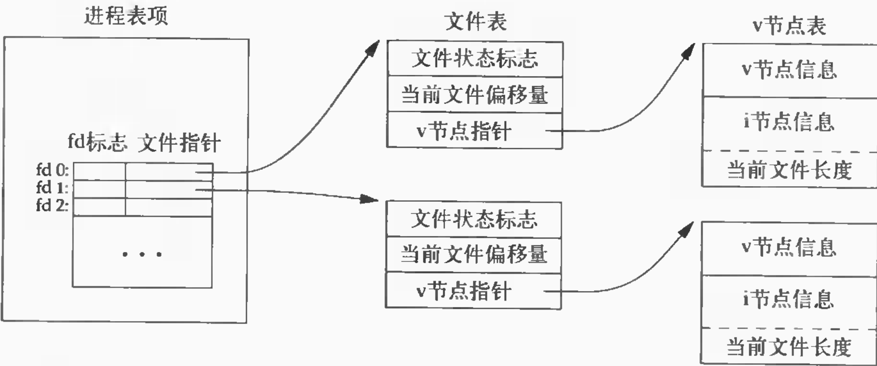


图3-1 打开文件的内核数据结构

> 创建v节点结构的目的是对在一个计算机系统上的多文件系统类型提供支持。这一工作是由Peter Weinberger（贝尔实验室）和Bill Joy（Sun公司）分别独立完成的。Sun称此种文件系统为虚拟文件系统（Virtual File System），称与文件系统类型无关的i节点部分为v节点[Kleiman 1986]。当各个制造商的实现增加了对Sun的网络文件系统（NFS）的支持时，它们都广泛采用了v节点结构。在BSD系列中首先提供v节点的是4.3BSD Reno版本，其中加入了NFS。
>
> 在SVR4中，v节点代换了SVR3中与文件系统类型无关的i节点结构。Solaris是从SVR4发展而来的，它也使用v节点。
>
> Linux没有将相关数据结构分为i节点和v节点，而是采用了一个独立于文件系统的i节点和一个依赖于文件系统的i节点。

71

如果两个独立进程各自打开了同一个文件，则有图3-2中所示的安排。我们假定第一个进程在文件描述符3上打开该文件，而另一个进程则在文件描述符4上打开该文件。打开该文件的每个进程都得到一个文件表项，但对一个给定的文件只有一个v节点表项。每个进程都有自己的文件表项的一个理由是：这种安排使每个进程都有它自己的对该文件的当前偏移量。

给出了这些数据结构后，现在对前面所述的操作作进一步说明。

- 在完成每个`write`后，在文件表项中的当前文件偏移量即增加所写的字节数。如果这使

当前文件偏移量超过了当前文件长度，则在i节点表项中的当前文件长度被设置为当前文件偏移量（也就是该文件加长了）。

- 如果用O_APPEND标志打开了一个文件，则相应标志也被设置到文件表项的文件状态标志中。每次对这种具有添写标志的文件执行写操作时，在文件表项中的当前文件偏移量首先被设置为i节点表项中的文件长度。这就使得每次写的数据都添加到文件的当前尾端处。
- 若一个文件用lseek定位到文件当前的尾端，则文件表项中的当前文件偏移量被设置为i节点表项中的当前文件长度。（注意，这与用O_APPEND标志打开文件是不同的，详见3.11节。）

72

- lseek函数只修改文件表项中的当前文件偏移量，没有进行任何I/O操作。

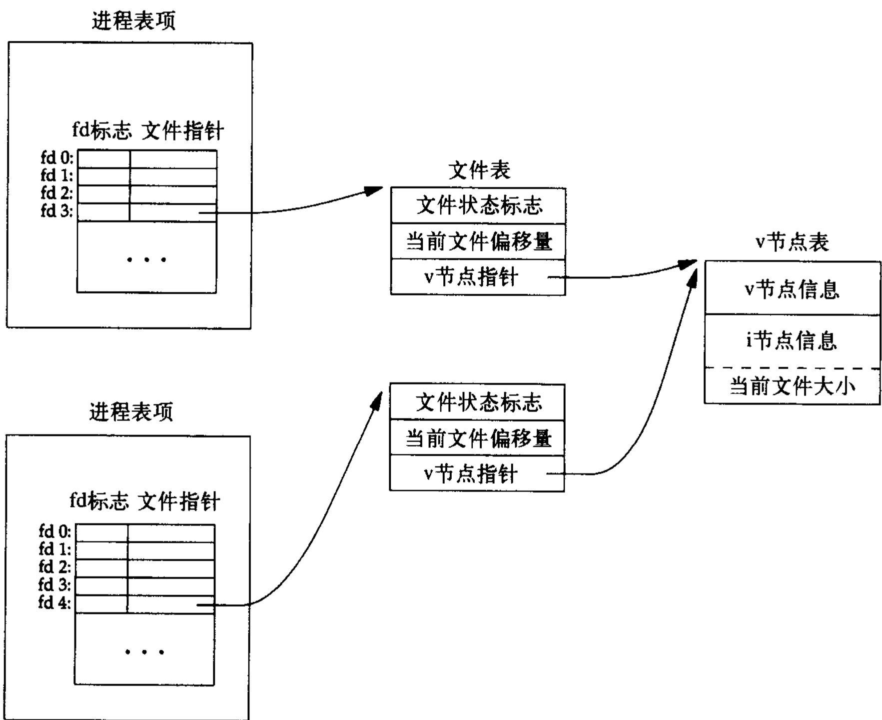


图3-2　两个独立进程各自打开同一个文件

可能有多个文件描述符项指向同一文件表项。在3.12节中讨论dup函数时，我们就能看到这一点。在fork后也会发生同样的情况，此时父、子进程对于每一个打开文件描述符共享同一个文件表项（见8.3节）。

注意，文件描述符标志和文件状态标志在作用域方面的区别，前者只用于一个进程的一个描述符，而后者则适用于指向该给定文件表项的任何进程中的所有描述符。在3.14节说明fcntl函数时，我们将会了解如何获取和修改文件描述符标志和文件状态标志。

本节上面所述的一切对于多个进程读同一文件都能正确工作。每个进程都有它自己的文件表项，其中也有它自己的当前文件偏移量。但是，当多个进程写同一个文件时，则可能产生预期不到的结果。为了说明如何避免这种情况，需要理解原子操作的概念。

73

3.11 原子操作

1. 添写至一个文件

考虑一个进程，它要将数据添加到一个文件尾端。早期的UNIX系统版本并不支持open的O_APPEND选项，所以程序被编写成下列形式：

```
if (lseek(fd, 0L, 2) < 0)          /* position to EOF */
   err_sys("lseek error");
if (write(fd, buf, 100) != 100)    /* and write */
   err_sys("write error");
```

对单个进程而言，这段程序能正常工作，但若有多个进程同时使用这种方法将数据添加到同一文件，则会产生问题。（例如，若此程序由多个进程同时执行，各自将消息添加到一个日志文件中，就会产生这种情况。）

假定有两个独立的进程A和B都对同一文件进行添加操作。每个进程都已打开了该文件，但未使用O_APPEND标志。此时，各数据结构之间的关系如图3-2中所示。每个进程都有它自己的文件表项，但是共享一个v节点表项。假定进程A调用了lseek，它将进程A的该文件当前偏移量设置为1500字节（当前文件尾端处）。然后内核切换进程使进程B运行。进程B执行lseek，也将其对该文件的当前偏移量设置为1500字节（当前文件尾端处）。然后B调用write，它将B的该文件当前文件偏移量增至1600。因为该文件的长度已经增加了，所以内核对v节点中的当前文件长度更新为1600。然后，内核又进行进程切换使进程A恢复运行。当A调用write时，就从其当前文件偏移量（1500字节）处将数据写到文件中去。这样也就代换了进程B刚写到该文件中的数据。

问题出在逻辑操作“定位到文件尾端处，然后写”上，它使用了两个分开的函数调用。解决问题的方法是使这两个操作对于其他进程而言成为一个原子操作。任何一个需要多个函数调用的操作都不可能是原子操作，因为在两个函数调用之间，内核有可能会临时挂起该进程（正如我们前面所假定的）。

UNIX系统提供了一种方法使这种操作成为原子操作，该方法是在打开文件时设置O_APPEND标志。正如前一节中所述，这就使内核每次对这种文件进行写之前，都将进程的当前偏移量设置到该文件的尾端处，于是在每次写之前就不再需要调用lseek。

2. pread和pwrite函数

Single UNIX Specification包括了XSI扩展，该扩展允许原子性地定位搜索（seek）和执行I/O。pread和pwrite就是这种扩展。

74

```
#include <unistd.h>

ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);
```

返回值：读到的字节数，若已到文件结尾则返回0，若出错则返回-1

```
ssize_t pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
```

返回值：若成功则返回已写的字节数，若出错则返回-1

调用pread相当于顺序调用lseek和read，但是pread又与这种顺序调用有下列重要区别：

- 调用pread时，无法中断其定位和读操作。

• 不更新文件指针。

调用pwrite相当于顺序调用lseek和write，但也与它们有类似的区别。

3. 创建一个文件

在对open函数的O_CREAT和O_EXCL选项进行说明时，我们已见到另一个有关原子操作的例子。当同时指定这两个选项，而该文件又已经存在时，open将失败。我们曾提及检查该文件是否存在以及创建该文件这两个操作是作为一个原子操作执行的。如果没有这样一个原子操作，那么可能会编写下列程序段：

```
if ((fd = open(pathname, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(pathname, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

如果在open和creat之间，另一个进程创建了该文件，那么就会引起问题。例如，若在这两个函数调用之间，另一个进程创建了该文件，并且写进了一些数据，然后，原先的进程执行这段程序中的creat，这时，刚由另一个进程写上去的数据就会被擦去。如若将这两者合并在一个原子操作中，这种问题也就不会产生。

一般而言，原子操作（atomic operation）指的是由多步组成的操作。如果该操作原子地执行，则要么执行完所有步骤，要么一步也不执行，不可能只执行所有步骤的一个子集。在4.15节论述link函数以及在14.3节中说明记录锁时，还将讨论原子操作。

75

3.12 dup和dup2函数

下面两个函数都可用来复制一个现存的文件描述符：

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

两函数的返回值：若成功则返回新的文件描述符，若出错则返回-1

由dup返回的新文件描述符一定是当前可用文件描述符中的最小数值。用dup2则可以用*filedes2*参数指定新描述符的数值。如果*filedes2*已经打开，则先将其关闭。如若*filedes*等于*filedes2*，则dup2返回*filedes2*，而不关闭它。

这些函数返回的新文件描述符与参数*filedes*共享同一个文件表项。图3-3显示了这种情况。

在此图中，我们假定进程执行了：

```
newfd = dup(1);
```

当此函数开始执行时，假定下一个可用的描述符是3（这是非常可能的，因为0、1和2由shell打开）。因为两个描述符指向同一文件表项，所以它们共享同一文件状态标志（读、写、添写等）以及同一当前文件偏移量。

每个文件描述符都有它自己的一套文件描述符标志。正如我们将在下一节中说明的那样，新描述符的执行时关闭（close-on-exec）标志总是由dup函数清除。

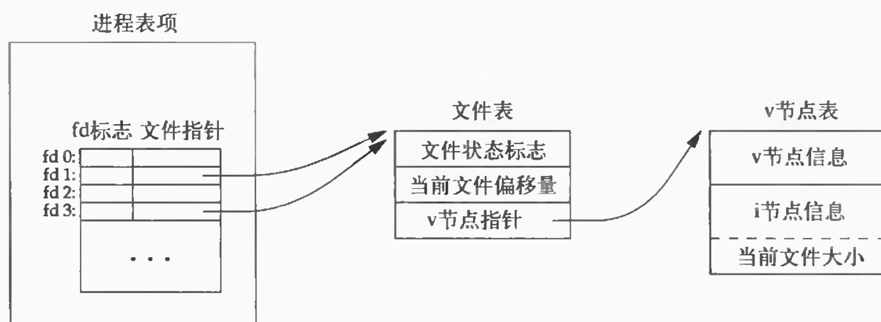


图3-3 执行dup(1)后的内核数据结构

复制一个描述符的另一种方法是使用fcntl函数，3.14节将对该函数进行说明。实际上，调用

```
dup(filedes);
```

等效于

```
fcntl(filedes, F_DUPFD, 0);
```

76

而调用

```
dup2(filedes, filedes2);
```

等效于

```
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

在后一种情况下，dup2并不完全等同于close加上fcntl。它们之间的区别是：

(1) dup2是一个原子操作，而close及fcntl则包括两个函数调用。有可能在close和fcntl之间插入执行信号捕获函数，它可能修改文件描述符（第10章将说明信号）。

(2) dup2和fcntl有某些不同的errno。

> dup2系统调用起源于V7，然后传播至所有BSD版本。而复制文件描述符的fcntl方法则首先由系统III使用，然后由系统V继续采用。SVR3.2选用了dup2函数，4.2BSD则选用了fcntl函数及F_DUPFD功能。POSIX.1要求兼有dup2及fcntl的F_DUPFD功能。

3.13 sync、fsync和fdatasync函数

传统的UNIX实现在内核中设有缓冲区高速缓存或页面高速缓存，大多数磁盘I/O都通过缓冲进行。当将数据写入文件时，内核通常先将该数据复制到其中一个缓冲区中，如果该缓冲区尚未写满，则并不将其排入输出队列，而是等待其写满或者当内核需要重用该缓冲区以便存放其他磁盘块数据时，再将该缓冲排入输出队列，然后待其到达队首时，才进行实际的I/O操作。这种输出方式被称为延迟写（delayed write）（Bach [1986]第3章详细讨论了缓冲区高速缓存）。

延迟写减少了磁盘读写次数，但是却降低了文件内容的更新速度，使得欲写到文件中的数据在一段时间内并没有写到磁盘上。当系统发生故障时，这种延迟可能造成文件更新内容的丢失。为了保证磁盘上实际文件系统与缓冲区高速缓存中内容的一致性，UNIX系统提供了sync、fsync和fdatasync三个函数。

```
#include <unistd.h>

int fsync(int filedes);

int fdatasync(int filedes);

                                    返回值：若成功则返回0，若出错则返回-1

void sync(void);
```

sync函数只是将所有修改过的块缓冲区排入写队列，然后就返回，它并不等待实际写磁盘操作结束。

通常称为update的系统守护进程会周期性地（一般每隔30秒）调用sync函数。这就保证了定期冲洗内核的块缓冲区。命令sync(1)也调用sync函数。

77

fsync函数只对由文件描述符*filedes*指定的单一文件起作用，并且等待写磁盘操作结束，然后返回。fsync可用于数据库这样的应用程序，这种应用程序需要确保将修改过的块立即写到磁盘上。

fdatasync函数类似于fsync，但它只影响文件的数据部分。而除数据外，fsync还会同步更新文件的属性。

> 本书说明的所有四种平台都支持sync和fsync函数。但是，FreeBSD 5.2.1和Mac OS X 10.3并不支持fdatasync。

3.14 fcntl函数

fcntl函数可以改变已打开的文件的性质。

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* int arg */ );

                          返回值：若成功则依赖于cmd（见下），若出错则返回-1
```

在本节的各实例中，第三个参数总是一个整数，与上面所示函数原型中的注释部分相对应。但是在14.3节说明记录锁时，第三个参数则是指向一个结构的指针。

fcntl函数有5种功能：

(1) 复制一个现有的描述符（*cmd*=F_DUPFD）。

(2) 获得/设置文件描述符标记（*cmd* = F_GETFD或F_SETFD）。

(3) 获得/设置文件状态标志（*cmd* = F_GETFL或F_SETFL）。

(4) 获得/设置异步I/O所有权（*cmd* = F_GETOWN或F_SETOWN）。

(5) 获得/设置记录锁（*cmd* = F_GETLK、F_SETLK或F_SETLKW）。

我们先说明这10种*cmd*值中的前7种（14.3节说明后3种，它们都与记录锁有关）。我们将涉及与进程表项中各文件描述符相关联的文件描述符标志，以及每个文件表项中的文件状态标志（见图3-1）。

F_DUPFD　复制文件描述符*filedes*。新文件描述符作为函数值返回。它是尚未打开的各描述符中大于或等于第三个参数值（取为整型值）中各值的最小值。新描述符与*filedes*共享同一文件表项（见图3-3）。但是，新描述符有它自己的一套文

件描述符标志，其FD_CLOEXEC文件描述符标志被清除（这表示该描述符在通过一个exec时仍保持有效，我们将在第8章对此进行讨论）。

F_GETFD　对应于*filedes*的文件描述符标志作为函数值返回。当前只定义了一个文件描述符标志FD_CLOEXEC。

78

F_SETFD　对于*filedes*设置文件描述符标志。新标志值按第三个参数（取为整型值）设置。

> 应当了解很多现有的涉及文件描述符标志的程序并不使用常量FD_CLOEXEC，而是将此标志设置为0（系统默认，在exec时不关闭）或1（在exec时关闭）。

F_GETFL　对应于*filedes*的文件状态标志作为函数值返回。在说明open函数时，已说明了文件状态标志。它们列于表3-3中。

表3-3　fcntl的文件状态标志

文件状态标志	说　明
O_RDONLY	只读打开
O_WRONLY	只写打开
O_RDWR	为读、写打开
O_APPEND	每次写时追加
O_NONBLOCK	非阻塞模式
O_SYNC	等待写完成（数据和属性）
O_DSYNC	等待写完成（仅数据）
O_RSYNC	同步读、写
O_FSYNC	等待写完成（仅FreeBSD 和 Mac OS X）
O_ASYNC	异步I/O（仅FreeBSD和 Mac OS X）

不幸的是，三个访问方式标志（O_RDONLY、O_WRONLY以及O_RDWR）并不各占1位（正如前述，这三种标志的值分别是0、1和2，由于历史原因。这三种值互斥——一个文件只能有这三种值之一）。因此首先必须用屏蔽字O_ACCMODE取得访问模式位，然后将结果与这三种值中的任一种作比较。

F_SETFL　将文件状态标志设置为第三个参数的值（取为整型值）。可以更改的几个标志是：O_APPEND、O_NONBLOCK、O_SYNC、O_DSYNC、O_RSYNC、O_FSYNC和O_ASYNC。

F_GETOWN　取当前接收SIGIO和SIGURG信号的进程ID或进程组ID。14.6.2节将论述这两种异步I/O信号。

F_SETOWN　设置接收SIGIO和SIGURG信号的进程ID或进程组ID。正的*arg*指定一个进程ID，负的*arg*表示等于*arg*绝对值的一个进程组ID。

fcntl的返回值与命令有关。如果出错，所有命令都返回−1，如果成功则返回某个其他值。下列四个命令有特定返回值：F_DUPFD、F_GETFD、F_GETFL以及F_GETOWN。第一个返回新的文件描述符，接下来的两个返回相应标志，最后一个返回一个正的进程ID或负的进程组ID。

79

实　例

程序清单3-4中的程序的第一个参数指定文件描述符，并对于该描述符打印其所选择的文件

标志说明。

程序清单3-4　对于指定的描述符打印文件标志

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int     val;

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
#if defined(O_SYNC)
    if (val & O_SYNC)
        printf(", synchronous writes");
#endif
#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

80

注意，我们使用了功能测试宏_POSIX_C_SOURCE，并且条件编译了POSIX.1中没有定义的文件访问标志。下面显示了从bash（Bourne-again shell）调用该程序时的几种情况。当使用不同shell时，结果会发生变化。

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
```

```
$ ./a.out 5 5<>temp.foo
read write
```

子句5<>temp.foo表示在文件描述符5上打开文件temp.foo以供读和写。 □

在修改文件描述符标志或文件状态标志时必须谨慎，先要取得现有的标志值，然后根据需要修改它，最后设置新标志值。不能只是执行F_SETFD或F_SETFL命令，这样会关闭以前设置的标志位。

程序清单3-5显示了一个对一个文件描述符设置一个或多个文件状态标志的函数。

程序清单3-5 对一个文件描述符打开一个或多个文件状态标志

```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

如果将中间的一条语句改为：

```
val &= ~flags;      /* turn flags off */
```

就构成另一个函数，我们称其为clr_fl，并将在后面某个例子中用到它。此语句使当前文件状态标志值val与flags的补码进行逻辑“与”运算。

81

如果在程序清单3-5的开始处，加上下面一行以调用set_fl，则打开了同步写标志。

```
set_fl(STDOUT_FILENO, O_SYNC);
```

这就使每次write都要等待，直至数据已写到磁盘上再返回。在UNIX系统中，通常write只是将数据排入队列，而实际的写磁盘操作则可能在以后的某个时刻进行。数据库系统很可能需要使用O_SYNC，这样一来，当它从write返回时就知道数据已确实写到了磁盘上，以免在系统崩溃时产生数据丢失。

程序运行时，设置O_SYNC标志会增加时钟时间。为了测试这一点，运行程序清单3-3，它从一个磁盘文件中将98.5 MB字节的数据复制到另一个文件。然后，在此程序中设置O_SYNC标志，使其完成上述同样的工作，以便将两者的结果进行比较。在使用ext2文件系统的Linux系统上执行上述操作，得到的结果见表3-4。

表3-4中的6行都是在BUFFSIZE为4096的情况下测量得到的。表3-2中的结果所测量的情况是读一个磁盘文件，然后写到/dev/null，所以没有磁盘输出。表3-4中的第2行对应于读一个磁盘文件，然后写到另一个磁盘文件中。这就是为什么表3-4中第1、2行有差别的原因。在写磁盘文件时，系统时间增加了，其原因是内核需要从进程中复制数据，并将数据排入队列以便

由磁盘驱动器将其写到磁盘上。当写至磁盘文件时，我们期望时钟时间也会增加，但在本测试中，它并未显著增加，这表明写操作将数据写到了系统高速缓存中，我们并没有测量将数据写到磁盘上的开销。

表3-4　用各种同步机制在Linux ext2环境中取得的计时结果

操　　作	用户CPU（秒）	系统CPU（秒）	时钟时间（秒）
取自表3-2中BUFFSIZE＝4 096的读时间	0.03	0.16	6.86
正常写到磁盘文件	0.02	0.30	6.87
设置O_SYNC后写到磁盘文件	0.03	0.30	6.83
写到磁盘后接着调用fdatasync	0.03	0.42	18.28
写到磁盘后接着调用fsync	0.03	0.37	17.95
在设置O_SYNC的条件下写到磁盘，接着调用fsync	0.05	0.44	17.95

当支持同步写时，系统时间和时钟时间应当会显著增加。从第3行可见，同步写所用的时间与延迟写所用的时间几乎相同。这意味着Linux ext2文件系统并未真正实现O_SYNC标志功能。第6行的时间值支持了我们的这种怀疑，其中显示，实施同步写，然后跟随fsync调用，这一操作序列所用的时间与写文件（未设置同步标志），然后接着执行fsync调用这一序列所用的时间（第5行）几乎相同。在同步写一个文件后，我们期望fsync调用不会产生影响。

表3-5显示了在Mac OS X 10.3上运行同样的测试所得到的计时结果。注意该计时结果与我们的期望相符：同步写较延迟写所消耗的时间增加了很多，而且在同步写后再调用函数fsync并不会产生测量上的显著差别。还要引起注意的是，在延迟写后增加一个fsync函数调用也不产生可测量的差别。其原因很可能是，当写新数据到某个文件中时，操作系统将以前写的数据冲洗到了磁盘上，所以在调用函数fsync时几乎就没有什么工作要做了。

82

表3-5　用各种同步机制在Mac OS X环境中取得的计时结果

操　　作	用户CPU（秒）	系统CPU（秒）	时钟时间（秒）
写至/dev/null	0.06	0.79	4.33
正常写到磁盘文件	0.05	3.56	14.40
设置O_FSYNC后写到磁盘文件	0.13	9.53	22.48
写到磁盘后接着调用fsync	0.11	3.31	14.12
在设置O_FSYNC的条件下写到磁盘，接着调用fsync	0.17	9.14	22.12

比较fsync和fdatasync与O_SYNC标志，fsync和fdatasync在我们需要时更新文件内容，O_SYNC标志则在我们每次写至文件时更新文件内容。　□

在本例中，我们看到了fcntl的必要性。我们的程序在一个描述符（标准输出）上进行操作，但是根本不知道由shell打开的相应文件的文件名。因为这是shell打开的，于是不能在打开时，按我们的要求设置O_SYNC标志。fcntl则允许仅知道打开文件描述符时可以修改其性质。在说明非阻塞管道时（15.2节），我们还将了解到，由于我们对管道所具有的知识只是其描述符，所以也需要使用fcntl的功能。

3.15　ioctl函数

ioctl函数是I/O操作的杂物箱。不能用本章中其他函数表示的I/O操作通常都能用ioctl

表示。终端I/O是ioctl的最大使用方面（第18章将介绍POSIX.1已经用一些新函数代替ioctl进行终端I/O操作）。

```
#include <unistd.h>      /* System V */
#include <sys/ioctl.h>   /* BSD and Linux */
#include <stropts.h>     /* XSI STREAMS */

int ioctl(int filedes, int request, ...);
```

返回值：若出错则返回-1，若成功则返回其他值

ioctl函数只是Single UNIX Specification标准的一个扩展，以便处理STREAMS设备[Rago 1993]。但是，UNIX系统实现用它进行很多杂项设备操作。有些实现甚至将它扩展到用于普通文件。

83

我们所示的原型对应于POSIX.1，FreeBSD 5.2.1和Mac OS X 10.3将第二个参数声明为unsigned long。因为第二个参数总是一个头文件中的defined名字，所以这种细节并没有什么影响。

对于ISO C原型，它用省略号表示其余参数。但是，通常只有另外一个参数，它常常是指向一个变量或结构的指针。

在此原型中，我们表示的只是ioctl函数本身所要求的头文件。通常，还要求另外的设备专用头文件。例如，除POSIX.1所说明的基本操作之外，终端I/O的ioctl命令都需要头文件<termios.h>。

每个设备驱动程序都可以定义它自己专用的一组ioctl命令。系统则为不同种类的设备提供通用的ioctl命令。表3-6总结了FreeBSD所支持的通用ioctl命令的一些类别。

表3-6 通用FreeBSD ioctl操作

类别	常量名	头文件	ioctl数
盘标号	DIOxxx	<sys/disklabel.h>	6
文件I/O	FIOxxx	<sys/filio.h>	9
磁带I/O	MTIOxxx	<sys/mtio.h>	11
套接字I/O	SIOxxx	<sys/sockio.h>	60
终端I/O	TIOxxx	<sys/ttycom.h>	44

磁带操作使我们可以在磁带上写一个文件结束标志、反绕磁带、越过指定个数的文件或记录等等，用本章中的其他函数（read、write、lseek等）都难以表示这些操作，所以，用ioctl是对这些设备进行操作的最容易方法。

在14.4节中说明STREAMS系统、18.12节中获取和设置终端窗口大小以及19.7节中论及伪终端的高级功能时，都将使用ioctl函数。

3.16 /dev/fd

较新的系统都提供名为/dev/fd的目录，其目录项是名为0、1、2等的文件。打开文件/dev/fd/*n*等效于复制描述符*n*（假定描述符*n*是打开的）。

/dev/fd这种特征是由Tom Duff开发的，它首先出现在Research UNIX System的第8版中，本书说明的所有四种系统都支持这种特征。它不是POSIX.1的组成部分。

84

在下列函数调用中：

```
fd = open("/dev/fd/0", mode);
```

大多数系统忽略它所指定的mode，而另外一些则要求mode必须是所涉及的文件（在这里则是标准输入）原先打开时所使用mode的子集。因为上面的打开等效于：

```
fd = dup(0);
```

所以描述符0和fd共享同一文件表项（见图3-3）。例如，若描述符0先前被打开为只读，那么我们也只能对fd进行读操作。即使系统忽略打开模式，并且下列调用成功：

```
fd = open("/dev/fd/0", O_RDWR);
```

我们仍然不能对fd进行写操作。

我们也可以用/dev/fd作为路径名参数调用creat，这与调用open时，用O_CREAT作为第2个参数作用相同。例如，若程序调用creat，并且路径名参数是/dev/fd/1等，那么该程序仍能工作。

某些系统提供路径名/dev/stdin、/dev/stdout和/dev/stderr。这些等效于/dev/fd/0、/dev/fd/1和/dev/fd/2。

/dev/fd文件主要由shell使用，它允许使用路径名作为调用参数的程序，能用处理其他路径名的相同方式处理标准输入和输出。例如，cat(1)程序对其命令行参数采取了一种特殊处理，它将单独的一个字符“-”解释为标准输入。例如：

```
filter file2 | cat file1 - file3 | lpr
```

首先cat读file1，接着读其标准输入（也就是filter file2命令的输出），然后读file3，如若支持/dev/fd，则可以删除cat对“-”的特殊处理，于是我们就可键入下列命令行：

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

在命令行中用“-”作为一个参数，特指标准输入或标准输出，这已由很多程序采用。但是这会带来一些问题，例如若用“-”指定第一个文件名，那么它看来就像指定了命令行中的一个选项。/dev/fd则提高了文件名参数的一致性，也更加清晰。

3.17 小结

本章说明了UNIX系统提供的基本I/O函数。因为read和write都在内核执行，所以称这些函数为不带缓冲的I/O函数。在只使用read和write情况下，我们观察了不同的I/O长度对读文件所需时间的影响。我们也观察了许多将写入的数据冲洗到磁盘上的方法，说明了它们对应用程序性能的影响。

85

在说明多个进程对同一文件进行添写操作以及多个进程创建同一文件时，本章介绍了原子操作。也介绍了内核用来共享打开文件信息的数据结构。在本书的稍后部分还将涉及这些数据结构。

我们还介绍了ioctl和fcntl函数。第14章还将使用这两个函数，将ioctl用于STREAMS I/O系统，将fcntl用于记录锁。

习题

3.1 当读/写磁盘文件时，本章中描述的函数是否有缓冲机制？请说明原因。

3.2 编写一个与3.12节中dup2功能相同的函数，要求不调用fcntl函数，并且要有正确的出错处理。

3.3 假设一个进程执行下面的3个函数调用：

```
fd1 = open(pathname, oflags);
fd2 = dup(fd1);
fd3 = open(pathname, oflags);
```

画出类似于图3-3的结果图。对fcntl作用于fd1来说，F_SETFD命令会影响哪一个文件描述符？F_SETFL呢？

3.4 在许多程序中都包含下面一段代码：

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
    close(fd);
```

为了说明if语句的必要性，假设fd是1，画出每次调用dup2时3个描述符项及相应的文件表项的变化情况。然后再画出fd为3的情况。

3.5 在Bourne shell、Bourne-again shell和Korn shell中，*digit*1 > &*digit*2表示要将描述符*digit*1重定向至描述符*digit*2的同一文件。请说明下面两条命令的区别。

```
./a.out > outfile 2>&1
./a.out 2>&1 > outfile
```

（提示：shell从左到右处理命令行。）

3.6 如果使用添加标志打开一个文件以便读、写，能否仍用lseek在任一位置开始读？能否用lseek更新文件中任一部分的数据？请编写一段程序以验证之。

86

第4章

文件和目录

4.1 引言

上一章我们说明了执行I/O操作的基本函数，其中的讨论是围绕普通文件I/O进行的——打开一个文件，读或写一个文件。本章将描述文件系统的其他特征和文件的性质。我们将从stat函数开始，逐个说明stat结构的每一个成员以了解文件的所有属性。在此过程中，我们将说明修改这些属性的各个函数（更改所有者、更改权限等），还将更详细地查看UNIX文件系统的结构以及符号链接。本章最后介绍了对目录进行操作的各个函数，并且开发了一个以降序遍历目录层次结构的函数。

4.2 stat、fstat和lstat函数

本章讨论的中心是三个stat函数以及它们所返回的信息。

```
#include <sys/stat.h>

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *restrict pathname, struct stat *restrict buf);
```

三个函数的返回值：若成功则返回0，若出错则返回-1

一旦给出*pathname*，stat函数就返回与此命名文件有关的信息结构。fstat函数获取已在描述符*filedes*上打开文件的有关信息。lstat函数类似于stat，但是当命名的文件是一个符号链接时，lstat返回该符号链接的有关信息，而不是由该符号链接引用文件的信息。（在4.21节中，当以降序遍历目录层次结构时，需要用到lstat。4.16节将更详细地说明符号链接。）

87

第二个参数*buf*是指针，它指向一个我们必须提供的结构。这些函数填写由*buf*指向的结构。该结构的实际定义可能随实现有所不同，但其基本形式是：

```
struct stat {
  mode_t    st_mode;    /* file type & mode (permissions) */
  ino_t     st_ino;     /* i-node number (serial number) */
  dev_t     st_dev;     /* device number (file system) */
  dev_t     st_rdev;    /* device number for special files */
  nlink_t   st_nlink;   /* number of links */
  uid_t     st_uid;     /* user ID of owner */
  gid_t     st_gid;     /* group ID of owner */
  off_t     st_size;    /* size in bytes, for regular files */
  time_t    st_atime;   /* time of last access */
  time_t    st_mtime;   /* time of last modification */
  time_t    st_ctime;   /* time of last file status change */
```

```
    blksize_t st_blksize; /* best I/O block size */
    blkcnt_t  st_blocks;  /* number of disk blocks allocated */
};
```

POSIX.1未要求st_rdev、st_blksize和st_blocks字段，Single UNIX Specification XSI扩展则定义了这些字段。

注意，该结构中的每一个成员都是基本系统数据类型（见2.8节）。我们将说明此结构的每个成员以了解文件属性。

使用stat函数最多的可能是ls -l命令，用其可以获得有关一个文件的所有信息。

4.3 文件类型

至今我们已介绍了两种不同的文件类型——普通文件和目录。UNIX系统的大多数文件是普通文件或目录，但是也有另外一些文件类型。文件类型包括如下几种：

(1) 普通文件（regular file）。这是最常用的文件类型，这种文件包含了某种形式的数据。至于这种数据是文本还是二进制数据对于UNIX内核而言并无区别。对普通文件内容的解释由处理该文件的应用程序进行。

一个值得注意的例外是二进制可执行文件。为了执行程序，内核必须理解其格式。所有二进制可执行文件都遵循一种格式，这种格式使内核能够确定程序文本和数据的加载位置。

(2) 目录文件（directory file）。这种文件包含了其他文件的名字以及指向与这些文件有关信息的指针。对一个目录文件具有读权限的任一进程都可以读该目录的内容，但只有内核可以直接写目录文件。进程必须使用本章说明的函数才能更改目录。

88

(3) 块特殊文件（block special file）。这种文件类型提供对设备（例如磁盘）带缓冲的访问，每次访问以固定长度为单位进行。

(4) 字符特殊文件（character special file）。这种文件类型提供对设备不带缓冲的访问，每次访问长度可变。系统中的所有设备要么是字符特殊文件，要么是块特殊文件。

(5) FIFO。这种类型文件用于进程间通信，有时也将其称为命名管道（named pipe）。15.5节将对其进行了说明。

(6) 套接字（socket）。这种文件类型用于进程间的网络通信。套接字也可用于在一台宿主机上进程之间的非网络通信。第16章将用套接字进行进程间的通信。

(7) 符号链接（symbolic link）。这种文件类型指向另一个文件。4.16节将更多地述及符号链接。

文件类型信息包含在stat结构的st_mode成员中。可以用表4-1中的宏确定文件类型。这些宏的参数都是stat结构中的st_mode成员。

表4-1 <sys/stat.h>中的文件类型宏

宏	文件类型
S_ISREG()	普通文件
S_ISDIR()	目录文件
S_ISCHR()	字符特殊文件
S_ISBLK()	块特殊文件
S_ISFIFO()	管道或FIFO
S_ISLNK()	符号链接
S_ISSOCK()	套接字

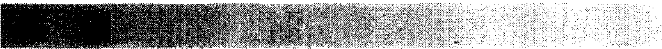
POSIX.1允许实现将进程间通信（IPC）对象（例如，消息队列和信号量等）表示为文件。表4-2中的宏可用来确定IPC对象的类型。这些宏与表4-1中的不同，它们的参数并非st_mode，而是指向stat结构的指针。

表4-2 <sys/stat.h>中的IPC类型宏

宏	对象的类型
S_TYPEISMQ()	消息队列
S_TYPEISSEM()	信号量
S_TYPEISSHM()	共享存储对象

消息队列、信号量以及共享存储对象等在第15章中讨论。但是，本书讨论的四种UNIX系统都不将这些对象表示为文件。

89



程序清单4-1中的程序取其命令行参数，然后针对每一个命令行参数打印其文件类型。

程序清单4-1 对每个命令行参数打印文件类型

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

程序清单4-1程序的示例输出是:

```
$ ./a.out /etc/passwd /etc /dav/initctl /dev/log /dev/tty \
> /dev/scsi/host0/bus0/target0/lun0/cd /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/initctl: fifo
/dev/log: socket
/dev/tty: character special
/dev/scsi/host0/bus0/target0/lun0/cd: block special
/dev/cdrom: symbolic link
```

90

(其中，在第一个命令行末端我们键入了一个反斜杠，通知shell要在下一行继续键入命令，然后，shell在下一行上用其第二提示符>提示我们。）我们特地使用了lstat函数而不是stat函数以便检测符号链接。如若使用了stat函数，则不会观察到符号链接。

为了在Linux系统上编译该程序，必须定义_GNU_SOURCE，这样就能包括S_ISSOCK宏的定义。 □

早期的UNIX系统版本并不提供S_ISxxx宏，于是就需要将st_mode与屏蔽字S_IFMT进行逻辑“与”运算，然后与名为S_IFxxx的常量相比较。大多数系统在文件<sys/stat.h>中定义了此屏蔽字和相关的常量。如若查看此文件，则可找到S_ISDIR宏定义为:

```
#define  S_ISDIR(mode)   (((mode) & S_IFMT) == S_IFDIR)
```

我们说过，普通文件是最主要的文件类型，但是观察一下在一个给定的系统中各种文件的比例是很有意思的。表4-3显示了在一个用作单用户工作站的Linux系统上的统计值和百分比。这些数据是由4.21节中的程序得到的。

表4-3 不同类型文件的统计值和百分比

文件类型	统计值	百分比（%）
普通文件	226 856	88.22
目录	23 017	8.95
符号链接	6 442	2.51
字符特殊	447	0.17
块特殊	312	0.12
套接字	69	0.03
FIFO	1	0.00

4.4 设置用户ID和设置组ID

与一个进程相关联的ID有6个或更多，它们示于表4-4中。

表4-4 与每个进程相关联的用户ID和组ID

ID	说明
实际用户ID 实际组ID	我们实际上是谁
有效用户ID 有效组ID 附加组ID	用于文件访问权限检查
保存的设置用户ID 保存的设置组ID	由exec函数保存

91

- 实际用户ID和实际组ID标识我们究竟是谁。这两个字段在登录时取自口令文件中的登录项。通常，在一个登录会话间这些值并不改变，但是超级用户进程有方法改变它们，8.10节将说明这些方法。
- 有效用户ID，有效组ID以及附加组ID决定了我们的文件访问权限，下一节将对此进行说明（我们已在1.8节中说明了附加组ID）。
- 保存的设置用户ID和保存的设置组ID在执行一个程序时包含了有效用户ID和有效组ID的副本，在8.1节中说明setuid函数时，将说明这两个保存值的作用。

在POSIX.1 2001版中，需要这些保存的ID。在早期POSIX版本中，它们是可选的。一个应用程序在编译时可测试常量_POSIX_SAVED_IDS，或在运行时以参数_SC_SAVED_IDS调用函数sysconf，以判断此实现是否支持这种特征。

通常，有效用户ID等于实际用户ID，有效组ID等于实际组ID。

每个文件都有一个所有者和组所有者，所有者由stat结构中的st_uid成员表示，组所有者则由st_gid成员表示。

当执行一个程序文件时，进程的有效用户ID通常就是实际用户ID，有效组ID通常是实际组ID。但是可以在文件模式字（st_mode）中设置一个特殊标志，其含义是“当执行此文件时，将进程的有效用户ID设置为文件所有者的用户ID（st_uid）”。与此相类似，在文件模式字中可以设置另一位，它使得将执行此文件的进程的有效组ID设置为文件的组所有者ID（st_gid）。在文件模式字中的这两位被称为设置用户ID（set-user-ID）位和设置组ID（set-group-ID）位。

例如，若文件所有者是超级用户，而且设置了该文件的设置用户ID位，然后当该程序由一个进程执行时，则该进程具有超级用户特权。不管执行此文件的进程的实际用户ID是什么，都进行这种处理。例如，UNIX程序passwd(1)允许任一用户改变其口令，该程序是一个设置用户ID程序。因为该程序应能将用户的新口令写入口令文件（一般是/etc/passwd或/etc/shadow）中，而只有超级用户才具有对该文件的写权限，所以需要使用设置用户ID特征。因为运行设置用户ID程序的进程通常得到额外的权限，所以编写这种程序时要特别谨慎。第8章将更详细地讨论这种类型的程序。

再返回到stat函数，设置用户ID位及设置组ID位都包含在st_mode值中。这两位可用常量S_ISUID和S_ISGID测试。

4.5 文件访问权限

st_mode值也包含了针对文件的访问权限位。当提及文件时，指的是前面所提到的任何类型的文件。所有文件类型（目录文件、字符特别文件等）都有访问权限（access permission）。很多人认为只有普通文件有访问权限，这是一种误解。

92

每个文件有9个访问权限位，可将它们分成三类，见表4-5。

在表4-5开头三行中，术语用户指的是文件所有者（owner）。chmod(1)命令用于修改这9个权限位。该命令允许我们用u表示用户（所有者），用g表示组，用o表示其他。有些书籍把这三种用户类型分别称为所有者、组和世界。这会造成混乱，因为chmod命令用o表示其他，而不是所有者。我们将使用术语用户、组和其他，以便与chmod命令一致。

表4-5中的三类访问权限（即读、写及执行）以各种方式由不同的函数使用。我们将这些不同的使用方式汇总在下面，当说明相关函数时，再进一步作讨论。

表4-5 9个访问权限位，取自<sys/stat.h>

st_mode屏蔽	意 义
S_IRUSR	用户-读
S_IWUSR	用户-写
S_IXUSR	用户-执行
S_IRGRP	组-读
S_IWGRP	组-写
S_IXGRP	组-执行
S_IROTH	其他-读
S_IWOTH	其他-写
S_IXOTH	其他-执行

- 第一个规则是，我们用名字打开任一类型的文件时，对该名字中包含的每一个目录，包括它可能隐含的当前工作目录都应具有执行权限。这就是为什么对于目录其执行权限位常被称为搜索位的原因。

  例如，为了打开文件/usr/include/stdio.h，需要对目录/、/usr和/usr/include具有执行权限。然后，需要具有对该文件本身的适当权限，这取决于以何种模式打开它（只读、读-写等）。

  如果当前目录是/usr/include，那么为了打开文件stdio.h，需要有对该工作目录的执行权限。这是隐含当前工作目录的一个实例。打开stdio.h文件与打开./stdio.h作用相同。

  注意，对于目录的读权限和执行权限的意义是不相同的。读权限允许我们读目录，获得在该目录中所有文件名的列表。当一个目录是我们要访问文件的路径名的一个组成部分时，对该目录的执行权限使我们可通过该目录（也就是搜索该目录，寻找一个特定的文件名）。

  引用隐含目录的另一个例子是，如果PATH环境变量（8.10节将对其进行说明）指定了一个我们不具有执行权限的目录，那么shell决不会在该目录下找到可执行文件。

93

- 对于一个文件的读权限决定了我们是否能够打开该文件进行读操作。这与open函数的O_RDONLY和O_RDWR标志相关。
- 对于一个文件的写权限决定了我们是否能够打开该文件进行写操作。这与open函数的O_WRONLY和O_RDWR标志相关。
- 为了在open函数中对一个文件指定O_TRUNC标志，必须对该文件具有写权限。
- 为了在一个目录中创建一个新文件，必须对该目录具有写权限和执行权限。
- 为了删除一个现有的文件，必须对包含该文件的目录具有写权限和执行权限。对该文件本身则不需要有读、写权限。
- 如果用6个exec函数（见8.10节）中的任何一个执行某个文件，都必须对该文件具有执行权限。该文件还必须是一个普通文件。

进程每次打开、创建或删除一个文件时，内核就进行文件访问权限测试，而这种测试可能涉及文件的所有者（st_uid和st_gid）、进程的有效ID（有效用户ID和有效组ID）以及进程的附加组ID（若支持的话）。两个所有者ID是文件的性质，而两个有效ID和附加组ID则是进程的性质。内核进行的测试是：

(1) 若进程的有效用户ID是0（超级用户），则允许访问。这给予了超级用户对整个文件系统进行处理的最充分的自由。

(2) 若进程的有效用户ID等于文件的所有者ID（也就是该进程拥有此文件），那么：若所有者适当的访问权限位被设置，则允许访问，否则拒绝访问。适当的访问权限位指的是，若进程为读而打开该文件，则用户读位应为1；若进程为写而打开该文件，则用户写位应为1；若进程将执行该文件，则用户执行位应为1。

(3) 若进程的有效组ID或进程的附加组ID之一等于文件的组ID，那么：若组适当的访问权限位被设置，则允许访问，否则拒绝访问。

(4) 若其他用户适当的访问权限位被设置，则允许访问，否则拒绝访问。

按顺序执行这四步。注意，如若进程拥有此文件（第2步），则按用户访问权限批准或拒绝该进程对文件的访问——不查看组访问权限。类似地，若进程并不拥有该文件，但进程属于某个适当的组，则按组访问权限批准或拒绝该进程对文件的访问——不查看其他用户的访问权限。 94

4.6 新文件和目录的所有权

在第3章中，当说明用open或creat创建新文件时，我们并没有说明赋予新文件的用户ID和组ID是什么。4.20节将说明mkdir函数，此时就会了解如何创建一个新目录。关于新目录的所有权规则与本节将说明的新文件所有权规则相同。

新文件的用户ID设置为进程的有效用户ID。关于组ID，POSIX.1允许实现选择下列之一作为新文件的组ID。

(1) 新文件的组ID可以是进程的有效组ID。

(2) 新文件的组ID可以是它所在目录的组ID。

> FreeBSD 5.2.1和Mac OS X 10.3总是使用目录的组ID作为新文件的组ID。
>
> Linux ext2和ext3文件系统允许基于文件系统在POSIX.1所允许的两种选项中选择一种，为此在mount(1)命令中使用了一个特殊标志。对Linux 2.4.22（使用适当的mount选项）和Solaris 9，新文件的组ID取决于它所在目录的设置组ID位是否设置。如果该目录的这一位已经设置，则将新文件的组ID设置为目录的组ID；否则将新文件的组ID设置为进程的有效组ID。

使用POSIX.1所允许的第二个选项（继承目录的组ID）使得在某个目录下创建的文件和目录都具有该目录的组ID。于是文件和目录的组所有权从该点向下传递。例如，在Linux的/var/spool/mail目录中就使用这种方法。

> 正如前面提到的，这种设置组所有权的方法对FreeBSD 5.2.1和Mac OS X 10.3是系统默认的，对Linux和Solaris则是可选的。在Linux 2.4.22和Solaris 9之下，必须使设置组ID位起作用。更进一步，为使这种方法能够正常工作，mkdir函数要自动地传递一个目录的设置组ID位（4.20节将说明mkdir就是这样做的）。

4.7 access函数

如前所述，当用open函数打开一个文件时，内核以进程的有效用户ID和有效组ID为基础执行其访问权限测试。有时，进程也希望按其实际用户ID和实际组ID来测试其访问能力。例如当一个进程使用设置用户ID或设置组ID特征作为另一个用户（或组）运行时，就可能会有这种需

要。即使一个进程可能已经因设置用户ID以超级用户权限运行，它仍可能想验证其实际用户能否访问一个给定的文件。access函数是按实际用户ID和实际组ID进行访问权限测试的。（该测试也分成四步，这与4.5节中所述的一样，但将有效改为实际。）

95

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

返回值：若成功则返回0，若出错则返回-1

其中，*mode*是表4-6中所列常量的按位或。

表4-6 access函数的*mode*常量，取自<unistd.h>

mode	说　明
R_OK	测试读权限
W_OK	测试写权限
X_OK	测试执行权限
F_OK	测试文件是否存在

程序清单4-2显示了access函数的使用方法。

程序清单4-2 access函数实例

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

下面是该程序的示例会话：

```
$ ls -l a.out
-rwxrwxr-x  1 sar        15945 Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls -l /etc/shadow
-r--------  1 root        1315 Jul 17  2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
```

```
open error for /etc/shadow: Permission denied
$ su                                          成为超级用户
Password:                                     输入超级用户口令
# chown root a.out                            将文件用户ID改为root
# chmod u+s a.out                             并打开设置用户ID位
# ls -l a.out                                 检查所有者和SUID位
-rwsrwxr-x  1 root       15945 Nov 30 12:10 a.out
# exit                                        回复为正常用户
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```


在本例中，设置用户ID程序可以确定实际用户不能读某个指定文件，而open函数却能打开该文件。 □

在上例及第8章中，我们有时要成为超级用户，以便演示某些功能是如何工作的。如果你使用多用户系统，但无超级用户权限，那么你就不能完整地重复这些实例。

4.8 umask函数

至此我们已说明了与每个文件相关联的9个访问权限位，在此基础上我们可以说明与每个进程相关联的文件模式创建屏蔽字。

umask函数为进程设置文件模式创建屏蔽字，并返回以前的值。（这是少数几个没有出错返回函数中的一个。）

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

返回值：以前的文件模式创建屏蔽字

其中，参数*cmask*是由表4-5中列出的9个常量（S_IRUSR、S_IWUSR等）中的若干个按位“或”构成的。

在进程创建一个新文件或新目录时，就一定会使用文件模式创建屏蔽字（回忆3.3节和3.4节，在那里我们说明了open和creat函数。这两个函数都有一个参数*mode*，它指定了新文件的访问权限位）。我们将在4.20节说明如何创建一个新目录。对于任何在文件模式创建屏蔽字中为1的位，在文件*mode*中的相应位则一定被关闭。

实 例

程序清单4-3中的程序创建了两个文件，创建第一个时，umask值为0，创建第二个时，umask值禁止所有组和其他用户的访问权限。

程序清单4-3 umask函数实例

```
#include "apue.h"
#include <fcntl.h>

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
```

```
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

97

若运行此程序可得如下结果，从中可见访问权限是如何设置的。

```
$ umask                            先打印当前文件模式创建屏蔽字
002
$ ./a.out
$ ls -l foo bar
-rw-------   1 sar                 0 Dec  7 21:20 bar
-rw-rw-rw-   1 sar                 0 Dec  7 21:20 foo
$ umask                            观察文件模式创建屏蔽字是否更改
002
```

□

UNIX系统的大多数用户从不处理他们的umask值。通常在登录时，由shell的启动文件设置一次，然后从不改变。尽管如此，当编写创建新文件的程序时，如果我们想确保指定的访问权限位已经激活，那么必须在进程运行时修改umask值。例如，如果我们想确保任何用户都能读文件，则应将umask设置为0。否则，当我们的进程运行时，有效的umask值可能关闭该权限位。

在前面的示例中，我们用shell的umask命令在运行程序的前、后打印文件模式创建屏蔽字。从中可见，更改进程的文件模式创建屏蔽字并不影响其父进程（常常是shell）的屏蔽字 。所有shell都有内置umask命令，我们可以用该命令设置或打印当前文件模式创建屏蔽字。

用户可以设置umask值以控制他们所创建文件的默认权限。该值表示成八进制数，一位代表一种要屏蔽的权限，这示于表4-7中。设置了相应位后，它所对应的权限就会被拒绝。常用的几种umask值是002、022和027，002阻止其他用户写你的文件，022阻止同组成员和其他用户写你的文件，027阻止同组成员写你的文件以及其他用户读、写或执行你的文件。

表4-7　umask文件访问权限位

屏蔽位	意　义
0400	用户读
0200	用户写
0100	用户执行
0040	组读
0020	组写
0010	组执行
0004	其他读
0002	其他写
0001	其他执行

98

Single UNIX Specification要求shell支持符号形式的umask命令。与八进制格式不同，符号格式指定许可的权限（即在文件创建屏蔽字中为0的位）而非拒绝的权限（即在文件创建屏蔽字中为1的位）。下面比较了两种格式的命令。

```
$ umask                  先打印当前文件模式创建屏蔽字
002
$ umask -S               打印符号形式
u=rwx,g=rwx,o=rx
$ umask 027              更改文件模式创建屏蔽字
$ umask -S               打印符号形式
u=rwx,g=rx,o=
```

4.9 chmod和fchmod函数

这两个函数使我们可以更改现有文件的访问权限。

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);
```

两个函数返回值：若成功则返回0，若出错则返回-1

chmod函数在指定的文件上进行操作，而fchmod函数则对已打开的文件进行操作。

为了改变一个文件的权限位，进程的有效用户ID必须等于文件的所有者ID，或者该进程必须具有超级用户权限。

参数*mode*是表4-8中所示常量的某种按位或运算构成的。

表4-8 chmod函数的*mode*常量，取自<sys/stat.h>

mode	说　明
S_ISUID	执行时设置用户ID
S_ISGID	执行时设置组ID
S_ISVTX	保存正文（粘住位）
S_IRWXU	用户（所有者）读、写和执行
S_IRUSR	用户（所有者）读
S_IWUSR	用户（所有者）写
S_IXUSR	用户（所有者）执行
S_IRWXG	组读、写和执行
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行
S_IRWXO	其他读、写和执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

99

注意，在表4-8中，有9项是取自表4-5中的9个文件访问权限位。我们另外加了6项，它们是两个设置ID常量（S_ISUID和S_ISGID）、保存正文常量（S_ISVTX），以及三个组合常量（S_IRWXU、S_IRWXG和S_IRWXO）。

保存-正文位（S_ISVTX）不是POSIX.1的一部分。在Single UNIX Specification中，它被定义在XSI扩展中。在下一节说明其目的。

实　例

为了演示umask函数，我们在前面运行了程序清单4-3中的程序，先让我们回忆文件foo和bar当时的最终状态：

```
$ ls -l foo bar
-rw-------   1 sar            0 Dec  7 21:20 bar
-rw-rw-rw-   1 sar            0 Dec  7 21:20 foo
```

程序清单4-4中的程序修改了这两个文件的模式。

程序清单4-4　chmod函数实例

```
#include "apue.h"

int
main(void)
{
    struct stat     statbuf;

    /* turn on set-group-ID and turn off group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */

    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");

    exit(0);
}
```

在运行程序清单4-4中的程序后，这两个文件的最终状态是：

```
$ ls -l foo bar
-rw-r--r--   1 sar            0 Dec  7 21:20 bar
-rw-rwSrw-   1 sar            0 Dec  7 21:20 foo
```

在本例中，不管文件bar的当前权限位如何，我们都将其权限设置为一绝对值。对于文件foo，我们相对于其当前状态设置权限。为此，先调用stat获得其当前权限，然后修改它。我们已显式地打开了设置组ID位、关闭了组执行位。注意，ls命令将组执行权限表示为S，它表示设置组ID位已设置，同时，组执行位则未设置。

100

> 在Solaris上，ls命令显示l而非S，这表示对该文件可以启用强制性文件或记录锁。这只能用于普通文件，14.3节更详细地讨论这一点。

最后也要注意到，在运行程序清单4-4中的程序后，ls命令列出的时间和日期并没有改变。在4.18节中，我们会了解到chmod函数更新的只是i节点最近一次被更改的时间。按系统默认方式，ls -l列出的是最后修改文件内容的时间。□

chmod函数在下列条件下自动清除两个权限位：

- Solaris等系统对用于普通文件的粘住位赋予了特殊含义，在这些系统上如果我们试图设置普通文件的粘住位（S_ISVTX），而且又没有超级用户特权，那么*mode*中的粘住位将自动被关闭（我们将在下一节说明粘住位）。这意味着只有超级用户才能设置普通文件的

粘住位。这样做的理由是防止不怀好意的用户设置粘住位，并由此降低系统性能。

> 在FreeBSD 5.2.1、Mac OS X 10.3和Solaris 9上，只有超级用户才能对普通文件设置粘住位。Linux 2.4.22对设置粘住位并无此种限制，其原因是，粘住位对Linux上的普通文件并无意义。虽然粘住位对FreeBSD和Mac OS X的普通文件也无意义，但是这两个系统阻止除超级用户以外的任何用户对普通文件设置该位。

- 新创建文件的组ID可能不是调用进程所属的组。回忆一下4.6节，新文件的组ID可能是父目录的组ID。特别地，如果新文件的组ID不等于进程的有效组ID或者进程附加组ID中的一个，以及进程没有超级用户特权，那么设置组ID位将自动被关闭。这就防止了用户创建一个设置组ID文件，而该文件是由并非该用户所属的组拥有的。

> FreeBSD 5.2.1、Linux 2.4.22、Mac OS X 10.3和Solaris 9增加了另一个安全性特征以试图阻止错误使用某些保护位。如果没有超级用户特权的进程写一个文件，则设置用户ID位和设置组ID位将自动被清除。如果不怀好意的用户找到一个他们可以写的设置组ID和设置用户ID文件，即使可以修改此文件，但他们也丢失了对该文件的特殊特权。

4.10 粘住位

S_ISVTX位有一段有趣的历史。在UNIX尚未使用分页技术的早期版本中，S_ISVTX位被称为粘住位（sticky bit）。如果一个可执行程序文件的这一位被设置了，那么在该程序第一次被执行并结束时，其程序正文部分的一个副本仍被保存在交换区。（程序的正文部分是机器指令部分。）这使得下次执行该程序时能较快地将其装入内存区。其原因是：交换区占用连续磁盘空间，可将它视为连续文件，而且一个程序的正文部分在交换区中也是连续存放的，而在一般的UNIX文件系统中，文件的各数据块很可能是随机存放的。对于常用的应用程序，例如文本编辑器和C编译器，我们常常设置它们所在文件的粘住位。自然，对于在交换区中可以同时存放的设置了粘住位的文件数是有一定限制的，以免过多占用交换区空间，但无论如何这是一个有用的技术。因为在系统再次自举前，文件的正文部分总是在交换区中，所以使用了名字“粘住”。后来的UNIX版本称它为保存正文位（saved-text bit），因此也就有了常量S_ISVTX。现今较新的UNIX系统大多数都配置有虚拟存储系统以及快速文件系统，所以不再需要使用这种技术。

101

现今的系统扩展了粘住位的使用范围，Single UNIX Specification允许针对目录设置粘住位。如果对一个目录设置了粘住位，则只有对该目录具有写权限的用户在满足下列条件之一的情况下，才能删除或更名该目录下的文件：

- 拥有此文件。
- 拥有此目录。
- 是超级用户。

目录/tmp和/var/spool/uucppublic是设置粘住位的典型候选者——任何用户都可在这两个目录中创建文件。任一用户（用户、组和其他）对这两个目录的权限通常都是读、写和执行。但是用户不应能删除或更名属于其他人的文件，为此在这两个目录的文件模式中都设置了粘住位。

> POSIX.1没有定义保存-正文位，Single UNIX Specification将它定义在XSI扩展部分。FreeBSD 5.2.1、Linux 2.4.22、Mac OS X 10.3和Solaris 9则支持这种特征。

在Solaris 9中，如果对普通文件设置了粘住位，那么它就具有特殊含义。在这种情况下，如若执行位都没有设置，那么操作系统就不会高速缓存文件内容。

4.11 chown、fchown和lchown函数

下面几个chown函数可用于更改文件的用户ID和组ID。

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int filedes, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

三个函数的返回值：若成功则返回0，若出错则返回-1

除了所引用的文件是符号链接以外，这三个函数的操作相似。在符号链接的情况下，lchown更改符号链接本身的所有者，而不是该符号链接所指向的文件。

Single UNIX Specification在其POSIX.1功能的扩展部分XSI中定义了lchown函数。因此预期所有的UNIX实现都会提供此函数。

如若两个参数owner或group中的任意一个是-1，则对应的ID不变。

102

基于BSD的系统一直规定只有超级用户才能更改一个文件的所有者。这样做的原因是防止用户改变其文件的所有者从而摆脱磁盘空间限额对他们的限制。系统V则允许任一用户更改他们所拥有的文件的所有者。

按照_POSIX_CHOWN_RESTRICTED的值，POSIX.1在这两种形式的操作中选用一种。

对于Solaris 9，此功能是个配置选项，其默认值是施加限制。而FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3则总对chown施加限制。

回忆2.6节，_POSIX_CHOWN_RESTRICTED常量可选地定义在头文件<unistd.h>中，而且总是可以用pathconf或fpathconf函数查询。此选项还与所引用的文件有关——可在每个文件系统基础上，使该选项起作用或不起作用。在下文中，如提及“若_POSIX_CHOWN_RESTRICTED起作用”，则表示这适用于我们正在谈及的文件，而不管该实际常量是否在头文件中定义。

若_POSIX_CHOWN_RESTRICTED对指定的文件起作用，则

(1) 只有超级用户进程能更改该文件的用户ID。

(2) 若满足下列条件，一个非超级用户进程就可以更改该文件的组ID：

  (a) 进程拥有此文件（其有效用户ID等于该文件的用户ID）。

  (b) 参数*owner*等于-1或文件的用户ID，并且参数*group*等于进程的有效组ID或进程的附加组ID之一。

这意味着，当_POSIX_CHOWN_RESTRICTED起作用时，不能更改其他用户文件的用户ID。你可以更改你所拥有的文件的组ID，但只能改到你所属的组。

如果这些函数由非超级用户进程调用，则在成功返回时，该文件的设置用户ID位和设置组ID位都会被清除。

4.12 文件长度

stat结构成员st_size表示以字节为单位的文件长度。此字段只对普通文件、目录文件和符号链接有意义。

> Solaris对管道也定义了文件长度，它表示可从该管道中读到的字节数，我们将在15.2节中讨论管道。

对于普通文件，其文件长度可以是0，在读这种文件时，将得到文件结束（end-of-file）指示。

对于目录，文件长度通常是一个数（例如16或512）的倍数，我们将在4.21节中说明读目录操作。

对于符号链接，文件长度是文件名中的实际字节数。例如，

```
lrwxrwxrwx  1 root            7 Sep 25 07:14 lib -> usr/lib
```

其中，文件长度7就是路径名usr/lib的长度（注意，因为符号链接文件长度总是由st_size指示，所以它并不包含通常C语言用作名字结尾的null字符）。

103

现今，大多数UNIX系统提供字段st_blksize和st_blocks。其中，第一个是对文件I/O较合适的块长度，第二个是所分配的实际512字节块数量。回忆3.9节，其中提到了当我们将st_blksize用于读操作时，读一个文件所需的时间量最少。为了效率的缘故，标准I/O库（我们将在第5章中说明）也试图一次读、写st_blksize个字节。

> 应当了解的是，不同的UNIX版本其st_blocks所用的单位可能不是512字节的块。使用此值并不是可移植的。

文件中的空洞

在3.6节中，我们提及普通文件可以包含空洞。在程序清单3-2中例证了这一点。空洞是由所设置的偏移量超过文件尾端，并写了某些数据后造成的。作为一个例子，考虑下列情况：

```
$ ls -l core
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
$ du -s core
272     core
```

文件core的长度刚好超过8 MB字节，而du命令则报告该文件所使用的磁盘空间总量是272个512字节块（139 264字节）（在很多BSD类系统上，du命令报告1024字节块块数，Solaris则报告512字节块块数）。很明显，此文件中有很多空洞。

正如我们在3.6节中提及的，对于没有写过的字节位置，read函数读到的字节是0。如果执行：

```
$ wc -c core
 8483248 core
```

> 由此可见，正常的I/O操作读整个文件长度（带-c选项的wc(1)命令用于统计文件中的字符（字节）数）。

如果使用实用程序（例如cat(1)）复制这种文件，那么所有这些空洞都会被填满，其中所有实际数据字节皆填写为0。

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
-rw-rw-r--  1 sar   8483248 Nov 18 12:27 core.copy
```

```
$ du -s core*
272     core
16592   core.copy
```

从中可见，新文件所用的实际字节数是8 495 104（512×16 592）。此长度与ls命令报告的长度不同，其原因是，文件系统使用了若干块以存放指向实际数据块的各个指针。

有兴趣的读者应当参阅Bach[1986]的4.2节，McKusick等[1996]的7.2节和7.3节（或104McKusick和Nevile-Neil[2005]的8.2节和8.3节）以及Mauro和McDougall[2001]的14.2节，以更详细地了解文件的物理结构。

4.13 文件截短

有时我们需要在文件尾端处截去一些数据以缩短文件。将一个文件清空为0是一个特例，在打开文件时使用O_TRUNC标志可以做到这一点。

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);

int ftruncate(int filedes, off_t length);
```

两个函数的返回值：若成功则返回0，若出错则返回-1

这两个函数将把现有的文件长度截短为*length*字节。如果该文件以前的长度大于*length*，则超过*length*以外的数据就不再能访问。如果以前的长度短于*length*，则其效果与系统有关。遵循XSI的系统将增加该文件的长度。若UNIX系统实现扩展了该文件，则在以前的文件尾端和新的文件尾端之间的数据将读作0（也就是可能在文件中创建了一个空洞）。

> ftruncate函数是POSIX.1的组成部分。在Single UNIX Specification中，truncate函数定义在POSIX.1功能的XSI扩展部分。
>
> 早于4.4BSD的系统只能用这三个函数截短一个文件——不能用它们扩展一个文件。
>
> Solaris对fcntl做了扩展，增加了F_FREESP命令，它允许释放一个文件中的任何一部分，而不只是文件尾端处的一部分。

程序清单13-2中的程序使用了ftruncate函数，以便在获得一个文件上的锁后，清空该文件。

4.14 文件系统

为了说明文件链接的概念，先要介绍UNIX文件系统的基本结构。同时，了解i节点和指向i节点的目录项之间的区别也是很有益的。

目前，正在使用的UNIX文件系统有多种实现。例如，Solaris支持多种不同类型的磁盘文件系统：传统的基于BSD的UNIX文件系统（称为UFS，UNIX file system），读、写DOS格式化软盘的文件系统（称为PCFS），以及读CD的文件系统（称为HSFS）。在表2-15中，我们已看到了不同类型文件系统之间的一个区别。UFS是以BSD快速文件系统为基础的。本节讨论该105文件系统。

我们可以把一个磁盘分成一个或多个分区。每个分区可以包含一个文件系统（见图4-1）。

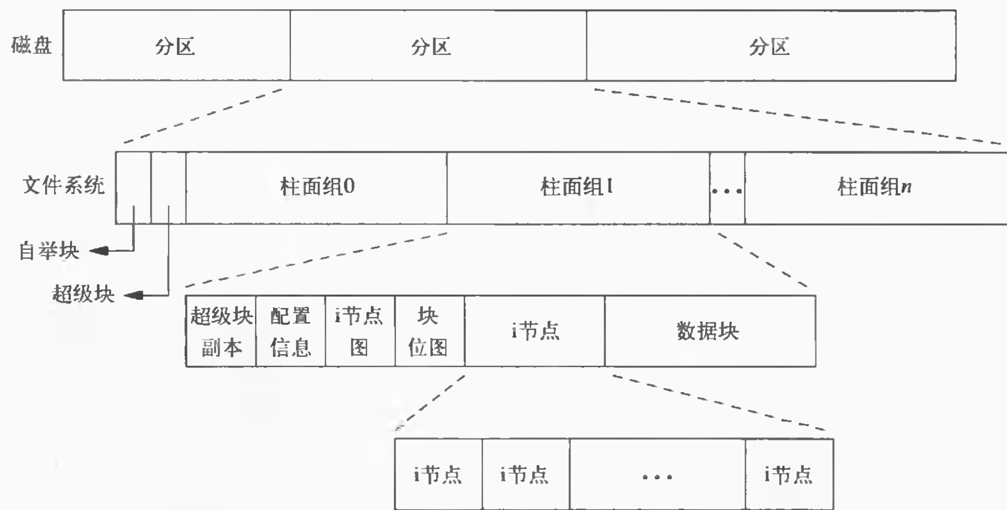


图4-1 磁盘、分区和文件系统

i节点是固定长度的记录项，它包含有关文件的大部分信息。

如果更仔细地观察一个柱面组的i节点和数据块部分，则可以看到图4-2中所示的情况。

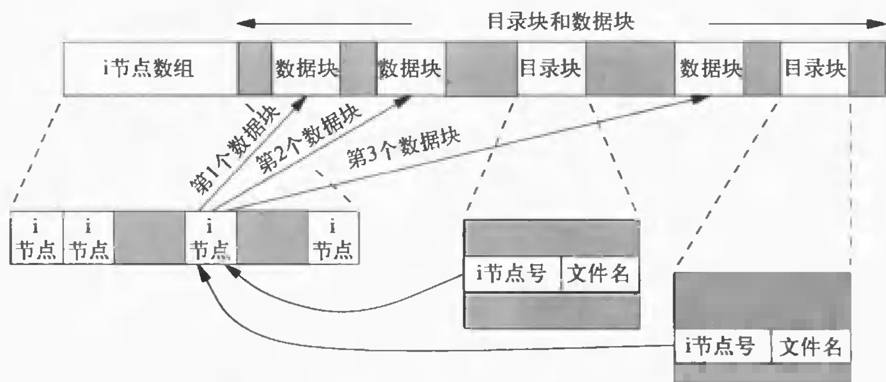


图4-2 较详细的柱面组的i节点和数据块

106

注意图4-2中的下列各点：

- 在图中有两个目录项指向同一i节点。每个i节点中都有一个链接计数，其值是指向该i节点的目录项数。只有当链接计数减少至0时，才可删除该文件（也就是可以释放该文件占用的数据块）。这就是为什么“解除对一个文件的链接”操作并不总是意味着“释放该文件占用的磁盘块”的原因。这也是为什么删除一个目录项的函数被称为unlink而不是delete的原因。在stat结构中，链接计数包含在st_nlink成员中，其基本系统数据类型是nlink_t。这种链接类型称为硬链接。回忆2.5.2节，其中，POSIX.1常量LINK_MAX指定了一个文件链接数的最大值。
- 另外一种链接类型称为符号链接（symbolic link）。对于这种链接，该文件的实际内容（在数据块中）包含了该符号链接所指向的文件的名字。在下例中：

```
lrwxrwxrwx  1 root       7 Sep 25 07:14 lib -> usr/lib
```

该目录项中的文件名是3字符的字符串lib，而在该文件中包含了7个数据字节usr/lib。该i节点中的文件类型是S_IFLNK，于是系统知道这是一个符号链接。

- i节点包含了大多数与文件有关的信息：文件类型、文件访问权限位、文件长度和指向该文件所占用的数据块的指针等等。stat结构中的大多数信息都取自i节点。只有两项数据存放在目录项中：文件名和i节点编号。i节点编号的数据类型是ino_t。
- 每个文件系统各自对它们的i节点进行编号，因此目录项中的i节点编号数指向同一文件系统中的相应i节点，不能使一个目录项指向另一个文件系统的i节点。这就是为什么ln(1)命令（构造一个指向一个现有文件的新目录项）不能跨越文件系统的原因。我们将在下一节说明link函数。
- 当在不更换文件系统情况下为一个文件更名时，该文件的实际内容并未移动，只需构造一个指向现有i节点的新目录项，并解除与旧目录项的链接。例如，为将文件/usr/lib/foo更名为/usr/foo，如果目录/usr/lib和/usr在同一文件系统中，则文件foo的内容无需移动。这就是mv(1)命令的通常操作方式。

我们说明了普通文件的链接计数概念，但是对于目录文件的链接计数字段又如何呢？假定我们在工作目录中构造了一个新目录：

```
$ mkdir testdir
```

107 图4-3显示了其结果。注意，该图显式地显示了.和..目录项。

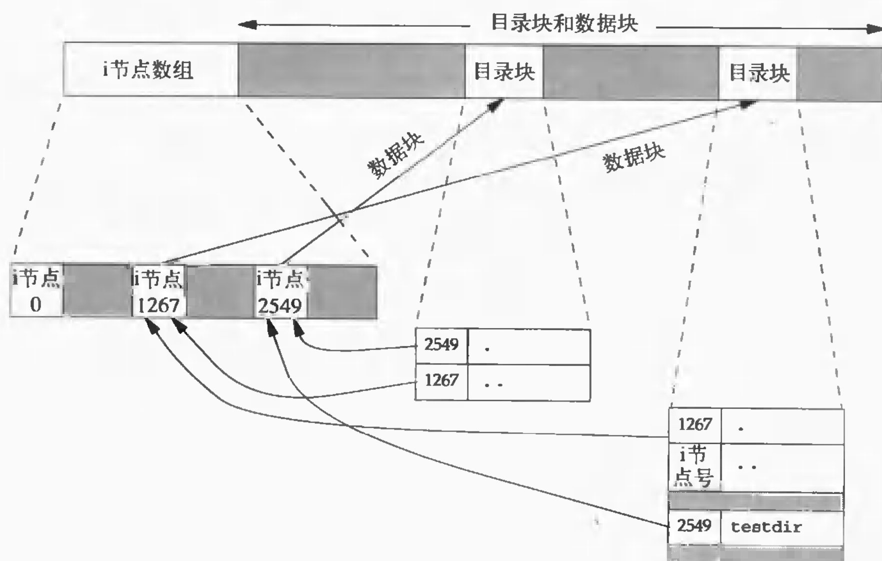


图4-3 创建了目录testdir后的示例柱形组

对于编号为2549的i节点，其类型字段表示它是一个目录，而链接计数为2。任何一个叶目录（不包含任何其他目录的目录）的链接计数总是2，数值2来自于命名该目录（testdir）的目录项以及在该目录中的.项。对于编号为1267的i节点，其类型字段表示它是一个目录，而其链接计数则大于或等于3。它大于或等于3的原因是，至少有三个目录项指向它：一个是命名它的目录项（在图4-3中没有表示出来），第二个是在该目录中的.项，第三个是在其子目录testdir中的.. 项。注意，父目录中的每一个子目录都会使该父目录的链接计数增1。

这种格式与UNIX文件系统的经典格式类似，在Bach[1986]一书的第4章中对此作了详细说

明。关于伯克利快速文件系统对此所作的更改请参阅McKusick等人[1996]的第7章或McKusick和Nevile-Neil[2005]中的第8章。关于UFS（伯克利快速文件系统的Solaris版）的详细情况，请参见Mauro和McDougall[2001]的第14章。

4.15 link、unlink、remove和rename函数

如上节所述，任何一个文件可以有多个目录项指向其i节点。创建一个指向现有文件的链接的方法是使用link函数。 108

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
```

返回值：若成功则返回0，若出错则返回-1

此函数创建一个新目录项*newpath*，它引用现有的文件*existingpath*。如若*newpath*已经存在，则返回出错。只创建*newpath*中的最后一个分量，路径中的其他部分应当已经存在。

创建新目录项以及增加链接计数应当是个原子操作（请回忆在3.11节中对原子操作的讨论）。虽然POSIX.1允许实现支持跨文件系统的链接，但是大多数实现要求这两个路径名在同一个文件系统中。如果实现支持创建指向一个目录的硬链接，那么也仅限于超级用户才可以这样做。其理由是这样做可能在文件系统中形成循环，大多数处理文件系统的实用程序都不能处理这种情况（4.16节将说明一个由符号链接引入的循环的例子）。因此很多文件系统实现不允许对于目录的硬链接。

为了删除一个现有的目录项，可以调用unlink函数。

```
#include <unistd.h>

int unlink(const char *pathname);
```

返回值：若成功则返回0，若出错则返回-1

此函数删除目录项，并将由*pathname*所引用文件的链接计数减1。如果还有指向该文件的其他链接，则仍可通过其他链接访问该文件的数据。如果出错，则不对该文件做任何更改。

我们在前面已经提及，为了解除对文件的链接，必须对包含该目录项的目录具有写和执行权限。正如4.10节所述，如果对该目录设置了粘住位，则对该目录必须具有写权限，并且具备下面三个条件之一：

- 拥有该文件。
- 拥有该目录。
- 具有超级用户特权。

只有当链接计数达到0时，该文件的内容才可被删除。另一个条件也会阻止删除文件的内容——只要有进程打开了该文件，其内容也不能删除。关闭一个文件时，内核首先检查打开该文件的进程数。如果该数达到0，然后内核检查其链接数，如果这个数也是0，那么就删除该文件的内容。 109

实例

程序清单4-5中的程序打开一个文件，然后解除对它的锁定。执行该程序的进程然后休眠15

秒钟，接着就终止。

程序清单4-5 打开一个文件，然后unlink它

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

运行该程序，其结果是：

```
$ ls -l tempfile                          查看文件大小
-rw-r-----  1 sar     413265408 Jan 21 07:14 tempfile
$ df /home                                检查可用磁盘空间
Filesystem    1K-blocks    Used  Available  Use%  Mounted on
/dev/hda4      11021440  1956332   9065108   18%  /home
$ ./a.out &                               在后台运行程序清单4-5
1364                                      shell打印其进程ID
$ file unlinked                           文件去链接
ls -l tempfile                            观察文件是否仍然存在
ls: tempfile: No such file or directory   目录项已删除
$ df /home                                检查可用磁盘空间有无变化
Filesystem    1K-blocks    Used  Available  Use%  Mounted on
/dev/hda4      11021440  1956332   9065108   18%  /home
$ done                                    程序执行结束，关闭所有打开的文件
df /home                                  现在，应当有更多可用的磁盘空间
Filesystem    1K-blocks    Used  Available  Use%  Mounted on
/dev/hda4      11021440  1552352   9469088   15%  /home
                                          现在，394.1 MB的磁盘空间可用
```
□

unlink的这种性质经常被程序用来确保即使是在该程序崩溃时，它所创建的临时文件也不会遗留下来。进程用open或creat创建一个文件，然后立即调用unlink。因为该文件仍旧是打开的，所以不会将其内容删除。只有当进程关闭该文件或终止时（在这种情况下，内核会关闭该进程打开的全部文件），该文件的内容才会被删除。

如果*pathname*是符号链接，那么unlink删除该符号链接，而不会删除由该链接所引用的文件。给出符号链接名情况下，没有一个函数能删除由该链接所引用的文件。

110

超级用户可以调用unlink，其参数*pathname*指定一个目录，但是通常应当使用rmdir函数，而不使用这种方式。我们将在4.20节中说明rmdir函数。

我们也可以用remove函数解除对一个文件或目录的链接。对于文件，remove的功能与unlink相同。对于目录，remove的功能与rmdir相同。

```
#include <stdio.h>

int remove(const char *pathname);
```

返回值：若成功则返回0，若出错则返回-1

ISO C指定remove函数删除一个文件，这更改了UNIX历来使用的名字unlink，其原因是实现C标准的大多数非UNIX系统并不支持文件链接。

文件或目录用rename函数更名。

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

返回值：若成功则返回0，若出错则返回-1

ISO C 对文件定义了此函数（C标准不处理目录）。POSIX.1扩展此定义，使其包含了目录和符号链接。

根据*oldname*是指文件还是目录，有几种情况要加以说明。我们也应说明如果*newname*已经存在将会发生什么。

(1) 如果*oldname*指的是一个文件而不是目录，那么为该文件或符号链接更名。在这种情况下，如果*newname*已存在，则它不能引用一个目录。如果*newname*已存在，而且不是一个目录，则先将该目录项删除然后将*oldname*更名为*newname*。对包含*oldname*的目录以及包含*newname*的目录，调用进程必须具有写权限，因为将更改这两个目录。

(2) 如若*oldname*指的是一个目录，那么为该目录更名。如果*newname*已存在，则它必须引用一个目录，而且该目录应当是空目录（空目录指的是该目录中只有.和..项）。如果*newname*存在（而且是一个空目录），则先将其删除，然后将*oldname*更名为*newname*。另外，当为一个目录更名时，*newname*不能包含*oldname*作为其路径前缀。例如，不能将/usr/foo更名为/usr/foo/testdir，因为旧名字（/usr/foo）是新名字的路径前缀，因而不能将其删除。

(3) 如若*oldname*或*newname*引用符号链接，则处理的是符号链接本身，而不是它所引用的文件。

111

(4) 作为一个特例，如果*oldname*和*newname*引用同一文件，则函数不做任何更改而成功返回。

如若*newname*已经存在，则调用进程对它需要有写权限（如同删除情况一样）。另外，调用进程将删除*oldname*目录项，并可能要创建*newname*目录项，所以它需要对包含*oldname*及包含*newname*的目录具有写和执行权限。

4.16 符号链接

符号链接是指向一个文件的间接指针，它与上一节所述的硬链接有所不同，硬链接直接指向文件的i节点。引入符号链接的原因是为了避开硬链接的一些限制：

- 硬链接通常要求链接和文件位于同一文件系统中。
- 只有超级用户才能创建指向目录的硬链接。

对符号链接以及它指向何种对象并无任何文件系统限制，任何用户都可创建指向目录的符号链接。符号链接一般用于将一个文件或整个目录结构移到系统中的另一个位置。

符号链接由4.2BSD引入，后来又得到SVR4的支持。

当使用以名字引用文件的函数时，应当了解该函数是否处理符号链接。也就是该函数是否跟随符号链接到达它所链接的文件。如若该函数具有处理符号链接的功能，则其路径名参数引

用由符号链接指向的文件。否则，路径名参数将引用链接本身，而不是该链接指向的文件。表4-9列出了本章中所说明的各个函数是否处理符号链接。在表4-9中没有列出mkdrir、mkinfo、mknod和rmdir这些函数，其原因是，当路径名是符号链接时，它们都出错返回。以文件描述符作为参数的一些函数（如fstat、fchmod等）也未在该表中列出，其原因是，对符号链接的处理是由返回文件描述符的函数（通常是open）进行的。chown是否跟随符号链接取决于实现。

在Linux 2.1.81之前的各版本中，chown并不跟随符号链接。从2.1.81版开始，chown跟随符号链接。FreeBSD 5.2.1和Mac OS X 10.3中的chown跟随符号链接。（在4.4BSD之前，chown不跟随符号链接，但在4.4BSD中这得到改变。）在Solaris 9中，chown也跟随符号链接。所有这些平台都实现了lchown，以改变符号链接自身的所有权。

表4-9的一个例外是，同时用O_CREAT和O_EXCL两者调用open函数。在此情况下，若路径名引用符号链接，open将出错返回，并将errno设置为EEXIST。这种处理方式的意图是堵塞一个安全性漏洞，使具有特权的进程不会被诱骗对不适当的文件进行写操作。

112

表4-9 各个函数对符号链接的处理

函数	不跟随符号链接	跟随符号链接
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

实例

使用符号链接可能在文件系统中引入循环。大多数查找路径名的函数在这种情况发生时都将返回值为ELOOP的errno。考虑下列命令序列：

```
$ mkdir foo                          创建一个新目录
$ touch foo/a                        创建0长文件
$ ln -s ../foo foo/testdir           创建符号链接
$ ls -l foo
total 0
```

```
-rw-r-----  1 sar          0 Jan 22 00:16 a
lrwxrwxrwx  1 sar          6 Jan 22 00:16 testdir -> ../foo
```

这创建了一个目录foo，它包含了一个名为a的文件以及一个指向foo的符号链接。在图4-4中显示了这种结果，图中以圆表示目录，以正方形表示一个文件。如果我们编写一段简单的程序，使用Solaris的标准函数ftw(3)以降序遍历文件结构，打印每个遇到的路径名，则其输出是：

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
```

（更多行，直至ftw出错返回，此时，errno值为ELOOP。）

113

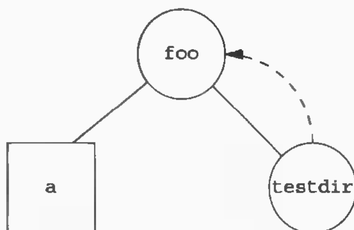


图4-4 构成循环的符号链接testdir

4.21节提供了我们自己的ftw函数版本，它用lstat代替stat以阻止它跟随符号链接。

> 注意，Linux的ftw函数使用lstat，所以它不显示这种程序运行行为。

这样一个循环是很容易消除的。因为unlink并不跟随符号链接，所以可以unlink文件foo/testdir。但是如果创建了一个构成这种循环的硬链接，那么就很难消除它。这就是为什么link函数不允许构造指向目录的硬链接的原因（除非进程具有超级用户特权）。

> 实际上，Rich Stevens在改写本节的原始版本时，在自己的系统上做了一个这样的实验。文件系统变得错误百出。正常的fsck(1)实用程序不能解决问题。为了修复文件系统，不得不使用了并不推荐使用的工具clri(8)和dcheck(8)。
>
> 需要对目录的硬链接是由来已久的，但是使用符号链接和mkdir函数，用户就不再需要创建指向目录的硬链接了。

用open打开文件时，如果传递给open函数的路径名指定了一个符号链接，那么open跟随此链接到达你所指定的文件。若此符号链接所指向的文件并不存在，则open返回出错，表示它不能打开该文件。这可能会使不熟悉符号链接的用户感到迷惑，例如：

```
$ ln -s /no/such/file myfile               创建符号链接
$ ls myfile
myfile                                     ls查到该文件
$ cat myfile                               试图查看该文件
cat: myfile: No such file or directory
$ ls -l myfile                             尝试-l选项
lrwxrwxrwx  1 sar       13 Jan 22 00:26 myfile -> /no/such/file
```

文件myfile存在，但cat却称没有这一文件。其原因是myfile是个符号链接，由该符号链接所指向的文件并不存在。ls命令的-l选项给我们两个提示：第一个字符是l，它表示这是一个符号链接，而->也表示这是一个符号链接。ls命令还有另一个选项-F，它在符号链接的文件名后加一个@符号，在未使用-l选项时，这可以帮助识别出符号链接。□

114

4.17 symlink和readlink函数

symlink函数创建一个符号链接。

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
```

返回值：若成功则返回0，若出错则返回-1

该函数创建了一个指向*actualpath*的新目录项*sympath*，在创建此符号链接时，并不要求*actualpath*已经存在（在上一节结束部分的例子中我们已经看到了这一点）。并且，*actualpath*和*sympath*并不需要位于同一文件系统中。

因为open函数跟随符号链接，所以需要有一种方法打开该链接本身，并读该链接中的名字。readlink函数提供了这种功能。

```
#include <unistd.h>

ssize_t readlink(const char* restrict pathname, char *restrict buf,
                 size_t bufsize);
```

返回值：若成功则返回读到的字节数，若出错则返回-1

此函数组合了open、read和close的所有操作。如果此函数成功执行，则它返回读入buf的字节数。在buf中返回的符号链接的内容不以null字符终止。

4.18 文件的时间

对每个文件保持有三个时间字段，它们的意义示于表4-10中。

表4-10　与每个文件相关的三个时间值

字　段	说　明	例　子	ls(1)选项
st_atime	文件数据的最后访问时间	read	-u
st_mtime	文件数据的最后修改时间	write	默认
st_ctime	i节点状态的最后更改时间	chmod、chown	-c

注意修改时间（st_mtime）和更改状态时间（st_ctime）之间的区别。修改时间是文件内容最后一次被修改的时间。更改状态时间是该文件的i节点最后一次被修改的时间。在本章中我们已说明了很多影响到i节点的操作，例如，更改文件的访问权限、用户ID、链接数等，但它们并没有更改文件的实际内容。因为i节点中的所有信息都是与文件的实际内容分开存放的，所以，除了文件数据修改时间以外，还需要更改状态的时间。

115

注意，系统并不保存对一个i节点的最后一次访问时间，所以access和stat函数并不更改这三个时间中的任一个。

系统管理员常常使用访问时间来删除在一定的时间范围内没有访问过的文件。典型的例子是删除在过去一周内没有访问过的名为a.out或core的文件。find (1) 命令常被用来进行这类操作。

修改时间和更改状态时间可被用来归档其内容已经被修改或其i节点已经被更改的那些文件。

ls命令按这三个时间值中的一个排序进行显示。按系统默认（用-l或-t选项调用时），它按文件的修改时间的先后排序显示。-u选项使其用访问时间排序，-c选项则使其用更改状态时间排序。

表4-11列出了我们已说明过的各种函数对这三个时间的作用。回忆4.14节中所述，目录是包含目录项（文件名和相关的i节点编号）的文件，增加、删除或修改目录项会影响到与其所在目录相关的三个时间。这就是在表4-11中包含两列的原因，其中一列是与该文件（或目录）相关的三个时间，另一列是与所引用的文件（或目录）的父目录相关的三个时间。例如，创建一个新文件会影响到包含此新文件的目录，也会影响该新文件的i节点。但是，读或写一个文件只会影响该文件的i节点，而对父目录则无影响（mkdir和rmdir函数将在4.20节中说明。utime函数将在下一节中说明。6个exec函数将在8.10节中讨论。第15章将说明mkfifo和pipe函数）。

表4-11 各种函数对访问、修改和更改状态时间的作用

函　数	引用的文件或目录			所引用文件或目录的父目录			节	备　注
	a	m	c	a	m	c		
chmod、fchmod			•				4.9	
chown、fchown			•				4.11	
creat	•	•	•		•	•	3.4	O_CREAT新文件
creat		•	•				3.4	O_TRUNC现有文件
exec	•						8.10	
lchown			•				4.11	
link			•		•	•	4.15	第二个参数的父目录
mkdir	•	•	•		•	•	4.20	
mkfifo	•	•	•		•	•	15.5	
open	•	•	•		•	•	3.3	O_CREAT新文件
open		•	•				3.3	O_TRUNC现有文件
pipe	•	•	•				15.2	
read	•						3.7	
remove			•		•	•	4.15	删除文件 =unlink
remove					•	•	4.15	删除目录=rmdir
rename			•		•	•	4.15	对于两个参数
rmdir					•	•	4.20	
truncate、ftruncate		•	•				4.13	
unlink			•		•	•	4.15	
utime	•	•	•				4.19	
write		•	•				3.8	

4.19 utime函数

一个文件的访问和修改时间可以用utime函数更改。

```
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *times);
```

返回值：若成功则返回0，若出错则返回−1

此函数所使用的数据结构是：

```
struct utimbuf {
  time_t  actime;   /* access time */
  time_t  modtime;  /* modification time */
}
```

116

此结构中的两个时间值是日历时间。如1.10节中所述，这是自1970年1月1日00:00:00以来国际标准时间所经过的秒数。

此函数的操作以及执行它所要求的特权取决于*times*参数是否是NULL。

- 如果*times*是一个空指针，则访问时间和修改时间两者都设置为当前时间。为了执行此操作必须满足下列两条件之一：进程的有效用户ID必须等于该文件的所有者ID；或者进程对该文件必须具有写权限。
- 如果*times*是非空指针，则访问时间和修改时间被设置为*times*所指向结构中的值。此时，进程的有效用户ID必须等于该文件的所有者ID，或者进程必须是一个超级用户进程。对文件只具有写权限是不够的。

注意，我们不能对更改状态时间st_ctime指定一个值，当调用utime函数时，此字段将被自动更新。

在某些UNIX系统版本中，touch(1)命令使用此函数。另外，标准归档程序tar(1)和cpio(1)可选地调用utime，以便将一个文件的时间设置为将它归档时保存的时间值。

117

程序清单4-6中的程序使用带O_TRUNC选项的open函数将文件长度截短为0，但并不更改其访问时间及修改时间。为了做到这一点，首先用stat函数得到这些时间，然后截短文件，最后再用utime函数复位这两个时间。

程序清单4-6 utime函数实例

```
#include "apue.h"
#include <fcntl.h>
#include <utime.h>

int
main(int argc, char *argv[])
{
    int             i, fd;
    struct stat     statbuf;
    struct utimbuf  timebuf;

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) {  /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
```

```
            continue;
        }
        close(fd);
        timebuf.actime  = statbuf.st_atime;
        timebuf.modtime = statbuf.st_mtime;
        if (utime(argv[i], &timebuf) < 0) {     /* reset times */
            err_ret("%s: utime error", argv[i]);
            continue;
        }
    }
    exit(0);
}
```

可以用以下脚本演示程序清单4-6中的程序：

```
$ ls -l changemod times           查看长度和最后修改时间
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ ls -lu changemod times          查看最后访问时间
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ dete                            打印当天日期
Thu Jan 22 06:55:17 EST 2004
$ ./a.out changemod times         运行图4-21程序
$ ls -l changemod times           检查结果
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lu changemod times          检查最后访问时间
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lc changemod times          检查更改状态时间
-rwxrwxr-x  1 sar       0 Jan 22 06:55 changemod
-rwxrwxr-x  1 sar       0 Jan 22 06:55 times
```

118

正如我们所预见的一样，最后修改时间和最后访问时间未变。但是，更改状态时间则更改为程序运行时的时间。 □

4.20 mkdir和rmdir函数

用mkdir函数创建目录，用rmdir函数删除目录。

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

返回值：若成功则返回0，若出错则返回−1

此函数创建一个新的空目录。其中，. 和.. 目录项是自动创建的。所指定的文件访问权限*mode*由进程的文件模式创建屏蔽字修改。

常见的错误是指定与文件相同的*mode*（只指定读、写权限）。但是，对于目录通常至少要设置1个执行权限位，以允许访问该目录中的文件名（见习题4.16）。

按照4.6节中讨论的规则，设置新目录的用户ID和组ID。

> Linux 2.4.22和Solaris 9也使新目录继承父目录的设置组ID位。这就使得在新目录中创建的文件将继承该目录的组ID。对于Linux，文件系统实现决定是否支持此特征。例如，ext2和ext3文件系统用mount(1)命令的一个选项来控制是否支持此特征。但是，Linux的UFS文件系统实现则是不可选择的，

新目录继承父目录的设置组ID位，这模仿了历史上BSD的实现，在BSD系统中，新目录的组ID是从父目录继承的。

基于BSD的系统并不要求继承此设置组ID位，因为不论设置组ID位如何，新创建的文件和目录总是继承父目录的组ID。因为FreeBSD 5.2.1和MAC OS X 10.3是基于4.4BSD的，它们不要求继承设置组ID位。在这些平台上，新创建的文件和目录总是继承父目录的组ID，这与设置组ID位无关。

早期的UNIX版本并没有mkdir函数，它是由4.2BSD和SVR3引入的。在早期版本中，进程要调用mknod函数以创建一个新目录。但是只有超级用户进程才能使用mknod函数。为了避免这一点，创建目录的命令mkdir(1)必须由根用户拥有，而且对它设置了设置用户ID位。进程为了创建一个目录，必须用system(3)函数调用mkdir(1)命令。

119

用rmdir函数可以删除一个空目录。空目录是只包含.和..这两项的目录。

```
#include <unistd.h>
int rmdir(const char *pathname);
```

返回值：若成功则返回0，若出错则返回-1

如果调用此函数使目录的链接计数成为0，并且也没有其他进程打开此目录，则释放由此目录占用的空间。如果在链接计数达到0时，有一个或几个进程打开了此目录，则在此函数返回前删除最后一个链接及. 和.. 项。另外，在此目录中不能再创建新文件。但是在最后一个进程关闭它之前并不释放此目录。（即使另一个进程打开该目录，它们在此目录下也不能执行其他操作。这样处理的原因是，为了使rmdir函数成功执行，该目录必须是空的。）

4.21 读目录

对某个目录具有访问权限的任一用户都可读该目录，但是，为了防止文件系统产生混乱，只有内核才能写目录。回忆4.5节，一个目录的写权限位和执行权限位决定了在该目录中能否创建新文件以及删除文件，它们并不表示能否写目录本身。

目录的实际格式依赖于UNIX系统，特别是其文件系统的具体设计和实现。早期的系统（例如V7）有一个比较简单的结构：每个目录项是16个字节，其中14个字节是文件名，2个字节是i节点编号。而对于4.2BSD而言，由于它允许相当长的文件名，所以每个目录项的长度是可变的。这就意味着读目录的程序与系统相关。为了简化这种情况，UNIX现在包含了一套与读目录有关的例程，它们是POSIX.1的一部分。很多实现阻止应用程序使用read函数读取目录的内容，从而进一步将应用程序与目录格式中与实现相关的细节隔离开。

```
#include <dirent.h>
DIR *opendir(const char *pathname);
```

返回值：若成功则返回指针，若出错则返回NULL

```
struct dirent *readdir(DIR *dp);
```

返回值：若成功则返回指针，若在目录结尾或出错则返回NULL

```
void rewinddir(DIR *dp);
int closedir(DIR *dp);
```

返回值：若成功则返回0，若出错则返回-1

```
long telldir(DIR *dp);
```

返回值：与dp关联的目录中的当前位置

120

```
void seekdir(DIR *dp, long loc);
```

telldir和seekdir函数不是基本POSIX.1标准的组成部分。它们是Single UNIX Specification中的XSI扩展，所以可以期望所有遵循UNIX系统的实现都会提供这两个函数。

回忆一下，在程序清单1-1中的程序中（ls命令的基本实现部分）使用了其中几个函数。

头文件<dirent.h>中定义的dirent结构与实现有关。几种典型的UNIX实现对此结构所作的定义至少包含下列两个成员：

```
struct dirent {
  ino_t  d_ino;                  /* i-node number */
  char   d_name[NAME_MAX + 1];  /* null-terminated filename */
}
```

POSIX.1并没有定义d_ino，因为这是一个实现特征，但在POSIX.1的XSI扩展中定义了d_ino。POSIX.1在此结构中只定义了d_name项。

注意，Solaris没有将NAME_MAX定义为一个常量——其值依赖于该目录所在的文件系统，并且通常可用fpathconf函数取得。NAME_MAX的常用值是255（见表2-12）。因为文件名是以null字符结束的，所以在头文件中如何定义数组d_name并无多大关系，数组大小并不表示文件名的长度。

DIR结构是一个内部结构，上述6个函数用这个内部结构保存当前正被读的目录的有关信息。其作用类似于FILE结构。FILE结构由标准I/O库维护（我们将在第5章中对它进行说明）。

由opendir返回的指向DIR结构的指针由另外5个函数使用。opendir执行初始化操作，使第一个readdir读目录中的第一个目录项。目录中各目录项的顺序与实现有关。它们通常并不按字母顺序排列。

实 例

我们将使用这些对目录进行操作的例程编写一个遍历文件层次结构的程序，其目的是得到如表4-3中所示的各种类型的文件数。程序清单4-7中的程序只有一个参数，它说明起点路径名，从该点开始递归降序遍历文件层次结构。Solaris提供了一个遍历此层次结构的函数ftw(3)，对于每一个文件它都会调用一个用户定义的函数。ftw函数的问题是：对于每一个文件，它都会调用stat函数，这就使程序跟随符号链接。例如，如果从root开始，并且有一个名为/lib的符号链接，它指向/usr/lib，则所有在目录/usr/lib中的文件都会计数两次。为了纠正这一点，Solaris提供了另一个函数nftw(3)，它具有一个停止跟随符号链接的选项。尽管可以使用nftw，但是为了说明目录例程的使用方法，我们还是编写了一个简单的文件遍历程序。 121

在Single UNIX Specification中，ftw和nftw都包含在基本POSIX.1标准的XSI扩展中。Solaris 9和Linux 2.4.22都包括了它们的实现。基于BSD的UNIX系统则有另一个函数fts(3)，它提供类似的功能。该函数在Free BSD 5.2.1、MAC OS X 10.3和Linux 2.4.22中是可用的。

程序清单4-7 递归降序遍历目录层次结构，并按文件类型计数

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* function type that is called for each filename */
typedef int Myfunc(const char *, const struct stat *, int);
```

```
static Myfunc   myfunc;
static int      myftw(char *, Myfunc *);
static int      dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int     ret;

    if (argc != 2)
        err_quit("usage:  ftw  <starting-pathname>");

    ret = myftw(argv[1], myfunc);       /* does it all */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1;       /* avoid divide by 0; print 0 for all counts */
    printf("regular files  = %7ld, %5.2f %%\n", nreg,
      nreg*100.0/ntot);
    printf("directories    = %7ld, %5.2f %%\n", ndir,
      ndir*100.0/ntot);
    printf("block special  = %7ld, %5.2f %%\n", nblk,
      nblk*100.0/ntot);
    printf("char special   = %7ld, %5.2f %%\n", nchr,
      nchr*100.0/ntot);
    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo,
      nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,
      nslink*100.0/ntot);
    printf("sockets        = %7ld, %5.2f %%\n", nsock,
      nsock*100.0/ntot);

    exit(ret);
}

/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */
#define FTW_F   1       /* file other than directory */
#define FTW_D   2       /* directory */
#define FTW_DNR 3       /* directory that can't be read */
#define FTW_NS  4       /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    int len;
    fullpath = path_alloc(&len);    /* malloc's for PATH_MAX+1 bytes */
                                    /* (Figure 2.15) */
    strncpy(fullpath, pathname, len);   /* protect against */
    fullpath[len-1] = 0;                /* buffer overrun */

    return(dopath(func));
}

/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return.  For a directory, we call ourself
```

122

```
 * recursively for each name in the directory.
 */
static int                    /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat     statbuf;
    struct dirent   *dirp;
    DIR             *dp;
    int             ret;
    char            *ptr;

    if (lstat(fullpath, &statbuf) < 0)  /* stat error */
        return(func(fullpath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0)  /* not a directory */
        return(func(fullpath, &statbuf, FTW_F));

    /*
     * It's a directory.  First call func() for the directory,
     * then process each filename in the directory.
     */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    ptr = fullpath + strlen(fullpath);  /* point to end of fullpath */
    *ptr++ = '/';
    *ptr = 0;

    if ((dp = opendir(fullpath)) == NULL)   /* can't read directory */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0  ||
            strcmp(dirp->d_name, "..") == 0)
                continue;       /* ignore dot and dot-dot */

        strcpy(ptr, dirp->d_name);  /* append name after slash */

        if ((ret = dopath(func)) != 0)      /* recursive */
            break;  /* time to leave */
    }
    ptr[-1] = 0;    /* erase everything from slash onwards */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);

    return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;     break;
        case S_IFBLK:   nblk++;     break;
        case S_IFCHR:   nchr++;     break;
        case S_IFIFO:   nfifo++;    break;
        case S_IFLNK:   nslink++;   break;
        case S_IFSOCK:  nsock++;    break;
        case S_IFDIR:
            err_dump("for S_IFDIR for %s", pathname);
                    /* directories should have type = FTW_D */
        }
```

123

```
            break;

        case FTW_D:
            ndir++;
            break;

        case FTW_DNR:
            err_ret("can't read directory %s", pathname);
            break;

        case FTW_NS:
            err_ret("stat error for %s", pathname);
            break;

        default:
            err_dump("unknown type %d for pathname %s", type, pathname);
        }

        return(0);
}
```

124

在程序中，我们提供了比所要求的更多的通用性，这样做的目的是为了具体说明ftw函数的应用。例如，函数myfunc总是返回0，即使调用它的函数准备处理非0返回也是如此。 □

关于降序遍历文件系统的更多信息，以及在很多标准UNIX命令（如find、ls、tar等）中使用这种技术的情况，请参阅Fowler、Korn和Vo[1989]。

4.22 chdir、fchdir和getcwd函数

每个进程都有一个当前工作目录，此目录是搜索所有相对路径名的起点（不以斜杠开始的路径名为相对路径名）。当用户登录到UNIX系统时，其当前工作目录通常是口令文件（/etc/passwd）中该用户登录项的第6个字段——用户的起始目录（home directory）。当前工作目录是进程的一个属性，起始目录则是登录名的一个属性。

进程通过调用chdir或fchdir函数可以更改当前工作目录。

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int filedes);
```

两个函数的返回值：若成功则返回0，若出错则返回-1

在这两个函数中，分别用*pathname*或打开文件描述符来指定新的当前工作目录。

fchdir不是基本POSIX.1规范的所属部分，在Single UNIX Specification中，它是XSI扩展部分。本书讨论的四种平台都支持此函数。

实　例

因为当前工作目录是进程的一个属性，所以它只影响调用chdir的进程本身，而不影响其他进程（我们将在第8章更详细地说明进程之间的关系）。这就意味着程序清单4-8中的程序并不会产生我们希望得到的结果。

程序清单4-8 chdir函数实例

```
#include "apue.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("chdir failed");
    printf("chdir to /tmp succeeded\n");
    exit(0);
}
```

125

如果编译程序清单4-8中的程序，并且调用其可执行目标代码文件mycd，则可以得到下列结果：

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

从中可以看出，执行mycd程序的shell的当前工作目录并没有改变，其原因是shell创建了一个子进程，由该子进程具体执行mycd程序。由此可见，为了改变shell进程自己的工作目录，shell应当直接调用chdir函数，为此cd命令的执行程序直接包含在shell程序中。 □

因为内核保持有当前工作目录的信息，所以我们应能取其当前值。不幸的是，内核为每个进程只保存指向该目录v节点的指针等目录本身的信息，并不保存该目录的完整路径名。

我们需要一个函数，它从当前工作目录（.目录项）开始，用..目录项找到其上一级的目录，然后读其目录项，直到该目录项中的i节点编号与工作目录i节点编号相同，这样就找到了其对应的文件名。按照这种方法，逐层上移，直到遇到根，这样就得到了当前工作目录完整的绝对路径名。很幸运，函数getcwd就提供了这种功能。

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

返回值：若成功则返回*buf*，若出错则返回NULL

向此函数传递两个参数，一个是缓冲地址*buf*，另一个是缓冲的长度*size*（单位：字节）。该缓冲必须有足够的长度以容纳绝对路径名再加上一个null终止字符，否则返回出错（请回忆2.5.5节中有关为最大长度路径名分配空间的讨论）。

> 某些getcwd的早期实现允许第一个参数*buf*为NULL。在这种情况下，此函数调用malloc动态地分配*size*字节数的空间。这不是POSIX.1或Single UNIX Specification的所属部分，应当避免使用。

实 例

程序清单4-9中的程序将工作目录更改至一个指定的目录，然后调用getcwd，最后打印该工作目录。如果运行该程序，则可得

```
$ ./a.out
```

```
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx  1 root  12 Jan 31 07:57 /usr/spool -> ../var/spool
```

126

程序清单4-9 getcwd函数实例

```
#include "apue.h"

int
main(void)
{
    char    *ptr;
    int     size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size);    /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

注意，chdir跟随符号链接（如表4-9中所示），但是当getcwd沿目录树上溯遇到/var/spool目录时，它并不了解该目录由符号链接/usr/spool所指向。这是符号链接的一种特性。

□

当一个应用程序需要在文件系统中返回到其工作的起点时，getcwd函数是有用的。在更换工作目录之前，我们可以调用getcwd函数先将其保存起来。在完成了处理后，就可将从getcwd获得的路径名作为调用参数传送给chdir，这样就返回到了文件系统中的起点。

fchdir函数向我们提供了一种完成此任务的便捷方法。在更换到文件系统中的不同位置前，无需调用getcwd函数，而是使用open打开当前工作目录，然后保存文件描述符。当希望回到原工作目录时，只要简单地将该文件描述符传递给fchdir。

4.23 设备特殊文件

st_dev和st_rdev这两个字段经常引起混淆，在18.9节编写ttyname函数时，需要使用这两个字段。有关规则很简单：

- 每个文件系统所在的存储设备都由其主、次设备号表示。设备号所用的数据类型是基本系统数据类型dev_t。主设备号标识设备驱动程序，有时编码为与其通信的外设板；次设备号标识特定的子设备。回忆图4-1，磁盘驱动器经常包含若干个文件系统。在同一磁盘驱动器上的各文件系统通常具有相同的主设备号，但它们的次设备号却不同。
- 我们通常可以使用两个宏即major和minor来访问主、次设备号，大多数实现都定义了这两个宏。这就意味着我们无需关心这两个数是如何存放在dev_t对象中的。

127

早期的系统用16位整型存放设备号：8位用于主设备号，8位用于次设备号。FreeBSD 5.2.1和Mac OS X 10.3使用32位整型，其中8位表示主设备号，24位表示次设备号。在32位系统上，Solaris 9用32位整型表示dev_t，其中14位用于主设备号，18位用于次设备号。在64位系统上，Solaris 9用64位整型数表示dev_t，各用其中的32位表示主设备号和次设备号。在Linux 2.4.22上，虽然dev_t是64位整型，但目前其主设备号和次设备号还只各用8位。

POSIX.1说明dev_t类型是存在的，但没有定义它包含什么，或如何取得其内容。大多数实现定义了宏major和minor，但在哪一个头文件中定义它们则与实现有关。基于BSD的UNIX系统将它们定义在<sys/types>中；Solaris将它们定义在<sys/mkdev.h>中；Linux将它们定义在<sys/sysmacros.h>中，而该头文件又包括在<sys/type.h>中。

- 系统中与每个文件名关联的st_dev值是文件系统的设备号，该文件系统包含了这一文件名以及与其对应的i节点。
- 只有字符特殊文件和块特殊文件才有st_rdev值。此值包含实际设备的设备号。

实例

程序清单4-10中的程序为每个命令行参数打印设备号，另外，若此参数引用的是字符特殊文件或块特殊文件，则还会打印该特殊文件的st_rdev值。

程序清单4-10 打印st_dev和st_rdev值

```
#include "apue.h"
#ifdef SOLARIS
#include <sys/mkdev.h>
#endif

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("stat error");
            continue;
        }

        printf("dev = %d/%d", major(buf.st_dev),  minor(buf.st_dev));
        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                    (S_ISCHR(buf.st_mode)) ? "character" : "block",
                    major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }

    exit(0);
}
```

128

运行此程序得到下面的结果：

```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 3/3
/home/sar: dev = 3/4
/dev/tty0: dev = 0/7 (character) rdev = 4/0
/dev/tty1: dev = 0/7 (character) rdev = 4/1
$ mount                           哪些目录安装在哪些设备上?
/dev/hda3 on / type ext2 (rw,noatime)
/dev/hda4 on /home type ext2 (rw,noatime)
```

```
$ ls -lL /dev/tty[01] /dev/hda[34]
brw-------  1 root      3,   3 Dec 31  1969 /dev/hda3
brw-------  1 root      3,   4 Dec 31  1969 /dev/hda4
crw-------  1 root      4,   0 Dec 31  1969 /dev/tty0
crw-------  1 root      4,   1 Jan 18 15:36 /dev/tty1
```

传递给该程序的前两个参数是目录（/和/home/sar），后两个是设备名/dev/tty[01]。（我们用shell正则表达式语言以缩短设备名。shell将扩展该字符串/dev/tty[01]为/dev/tty0/dev/tty1。）

这两个设备是字符特殊设备。从程序的输出可见，根目录和/home/sar目录的设备号不同，这表示它们位于不同的文件系统中。运行mount(1)命令证明了这一点。

然后用ls命令查看由mount命令报告的两个磁盘设备和两个终端设备。这两个磁盘设备是块特殊文件，而两个终端设备则是字符特殊文件。（通常，只有块特殊文件类型的设备才能包含随机访问文件系统，它们是：硬盘驱动器、软盘驱动器和CD-ROM等。UNIX的早期版本支持磁带存放文件系统，但这从未广泛使用过。）

注意，两个终端设备（st_dev）的文件名和i节点在设备0/7上（devfs伪文件系统，它实现了/dev文件系统），但是它们的实际设备号是4/0和4/1。 □

129

4.24　文件访问权限位小结

我们已经说明了所有文件访问权限位，其中某些位有多种用途。表4-12列出了所有这些权限位，以及它们对普通文件和目录文件的作用。

表4-12　文件访问权限位小结

常　量	说　明	对普通文件的影响	对目录的影响
S_ISUID	设置用户ID	执行时设置有效用户ID	（不使用）
S_ISGID	设置组ID	若组执行位设置，则执行时设置有效组ID，否则使强制性记录锁起作用（若支持）	将在目录中创建的新文件的组ID设置为目录的组ID
S_ISVTX	粘住位	在交换区保存程序正文（若支持）	限制在目录中删除和更名文件
S_IRUSR	用户读	许可用户读文件	许可用户读目录项
S_IWUSR	用户写	许可用户写文件	许可用户在目录中删除和创建文件
S_IXUSR	用户执行	许可用户执行文件	许可用户在目录中搜索给定路径名
S_IRGRP	组读	许可组读文件	许可组读目录项
S_IWGRP	组写	许可组写文件	许可组在目录中删除和创建文件
S_IXGRP	组执行	许可组执行文件	许可组在目录中搜索给定路径名
S_IROTH	其他读	许可其他读文件	许可其他读目录项
S_IWOTH	其他写	许可其他写文件	许可其他在目录中删除和创建文件
S_IXOTH	其他执行	许可其他执行文件	许可其他在目录中搜索给定路径名

最后9个常量分成3组，因为

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

4.25　小结

本章内容围绕stat函数，详细介绍了stat结构中的每一个成员。这使我们对UNIX文件的

各个属性都有所了解。对文件的所有属性以及操作文件的所有函数有完整的了解对UNIX编程是非常重要的。 130

习题

4.1 用stat函数替换程序清单4-1中的lstat函数，如若命令行参数之一是符号链接，那么会发生什么变化。

4.2 如果文件模式创建屏蔽字是777（八进制），结果会怎样？用shell的umask命令验证该结果。

4.3 关闭一个你所拥有文件的用户读权限，将导致不允许你访问自己的文件，对此进行验证。

4.4 创建文件foo和bar后，运行程序清单4-3中的程序，这将发生什么情况？

4.5 4.12节中讲到一个普通文件的大小可以为0，同时我们又知道st_size字段是为目录或符号链接定义的，那么目录和符号链接的长度是否可以为0？

4.6 编写一个类似cp(1)的程序，它复制包含空洞的文件，但不将字节0写到输出文件中去。

4.7 在4.12节ls命令的输出中，core和core.copy的访问权限不相同，如果创建两个文件时umask没有变，说明为什么会产生这种差别。

4.8 在运行程序清单4-5中的程序时，使用了df(1)命令来检查空闲的磁盘空间。为什么不使用du(1)命令？

4.9 表4-11中显示unlink函数会修改文件状态改变时间，这是怎样发生的？

4.10 4.21节中，系统对可打开文件数的限制对myftw函数会产生什么影响？

4.11 4.21节中的myftw从不改变其目录，对这种处理方法进行改动：每次遇到一个目录就用其调用chdir，这样每次调用lstat时就可以使用文件名而非路径名，处理完所有的目录项后执行chdir（".."）。比较这种版本的程序和正文中程序的运行时间。

4.12 每个进程都有一个根目录用于解析绝对路径名，可以通过chroot函数改变根目录。在手册中查阅此函数，说明这个函数什么时候有用。

4.13 如何使用utime函数只设置两个时间值中的一个？

4.14 有些版本的finger(1)命令输出“New mail received...”和”unread since...”，其中...表示相应的日期和时间。程序是如何决定这些日期和时间的？

4.15 用cpio(1)和tar(1)命令检查档案文件的格式（请参阅UNIX Programmer’s Manual 第5节中的说明）。三个时间值中哪几个是为每一个文件保存的？文件复原时，文件访问时间是什么？为什么？

4.16 UNIX对目录的深度有限制吗？编写一个程序循环，在每次循环中，创建目录，并将该目录更改为工作目录。确保叶节点的绝对路径名的长度大于系统的PATH_MAX限制。可以调用getcwd得到目录的路径名吗？标准UNIX工具是如何处理长路径名的？对目录可以使用tar或cpio命令归档目录吗？ 131

4.17 3.16节中描述了/dev/fd特征，如果每个用户都可以访问这些文件，则其访问权限必须为rw-rw-rw-。有些程序创建输出文件时，先删除该文件以确保该文件名不存在。

```
unlink(path);
if ((fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

讨论一下path是/dev/fd/1的情况。 132

第 5 章

标准I/O库

5.1 引言

本章说明标准I/O库。因为不仅在UNIX上，而且在很多操作系统上都实现了此库，所以它由ISO C标准说明。Single UNIX Specification对ISO C标准进行了扩展，定义了另外一些接口。

标准I/O库处理很多细节，例如缓冲区分配，以优化长度执行I/O等。这些处理使用户不必担心如何选择使用正确的块长度（如3.9节中所述）。这使得它便于用户使用，但是如果不较深入地了解I/O库函数的操作，也会带来一些问题。

> 标准I/O库是由Dennis Ritchie在1975年左右编写的。它是由Mike Lesk编写的可移植I/O库的主要修改版本。令人惊讶的是，经过30年后，对标准I/O库只做了极小的修改。

5.2 流和FILE对象

在第3章中，所有I/O函数都是针对文件描述符的。当打开一个文件时，即返回一个文件描述符，然后该文件描述符就用于后续的I/O操作。而对于标准I/O库，它们的操作则是围绕流（stream）进行的（请勿将标准I/O术语流与系统V的STREAMS I/O系统相混淆，STREAM I/O系统是系统V的组成部分，Single UNIX Specification则将其标准化为XSI STREAM选项）。当用标准I/O库打开或创建一个文件时，我们已使一个流与一个文件相关联。

133

对于ASCII字符集，一个字符用一个字节表示。对于国际字符集，一个字符可用多个字节表示。标准I/O文件流可用于单字节或多字节（“宽”）字符集。**流的定向**（streram's orientation）决定了所读、写的字符是单字节还是多字节的。当一个流最初被创建时，它并没有定向。如若在未定向的流上使用一个多字节I/O函数（见<wchar.h>），则将该流的定向设置为宽定向的。若在未定向的流上使用一个单字节I/O函数，则将该流的定向设置为字节定向的。只有两个函数可改变流的定向。freopen函数（参见5.5节）清除一个流的定向；fwide函数设置流的定向。

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *fp, int mode);
```

返回值：若流是宽定向的则返回正值，若流是字节定向的则返回负值，或者若流是未定向的则返回0

根据*mode*参数的不同值，fwide函数执行不同的工作：

- 如若*mode*参数值为负，fwide将试图使指定的流是字节定向的。
- 如若*mode*参数值为正，fwide将试图使指定的流是宽定向的。
- 如若*mode*参数值为0，fwide将不试图设置流的定向，但返回标识该流定向的值。

注意，fwide并不改变已定向流的定向。还应注意的是，fwide无出错返回。试想如若流是无效的，那么将发生什么呢？我们唯一可依靠的是，在调用fwide前先清除errno，从fwide返回时检查errno的值。在本书的其余部分，我们只涉及字节定向流。

当打开一个流时，标准I/O函数fopen返回一个指向FILE对象的指针。该对象通常是一个结构，它包含了标准I/O库为管理该流所需要的所有信息，包括：用于实际I/O的文件描述符、指向用于该流缓冲区的指针、缓冲区的长度、当前在缓冲区中的字符数以及出错标志等等。

应用程序没有必要检验FILE对象。为了引用一个流，需将FILE指针作为参数传递给每个标准I/O函数。在整本书中，我们称指向FILE对象的指针（类型为FILE*）为文件指针。

在本章中，我们在UNIX系统的环境中，说明标准I/O库。正如前述，此标准库已移植到除UNIX以外的很多其他操作系统中。但是为了说明该库实现的一些细节，我们将讨论其在UNIX系统上的典型实现。

134

5.3 标准输入、标准输出和标准出错

对一个进程预定义了三个流，并且这三个流可以自动地被进程使用，它们是：标准输入、标准输出和标准出错。这些流引用的文件与3.2节中提到的文件描述符STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO所引用的文件相同。

这三个标准I/O流通过预定义文件指针stdin、stdout和stderr加以引用。这三个文件指针同样定义在头文件<stdio.h>中。

5.4 缓冲

标准I/O库提供缓冲的目的是尽可能减少使用read和write调用的次数（见表3-2，其中显示了在不同缓冲区长度情况下，为执行I/O所需的CPU时间量）。它也对每个I/O流自动地进行缓冲管理，从而避免了应用程序需要考虑这一点所带来的麻烦。不幸的是，标准I/O库最令人迷惑的也是它的缓冲。

标准I/O提供了三种类型的缓冲：

(1) 全缓冲。这种情况下，在填满标准I/O缓冲区后才进行实际I/O操作。对于驻留在磁盘上的文件通常是由标准I/O库实施全缓冲的。在一个流上执行第一次I/O操作时，相关标准I/O函数通常调用malloc（见7.8节）获得需使用的缓冲区。

术语冲洗（flush）说明标准I/O缓冲区的写操作。缓冲区可由标准I/O例程自动冲洗（例如当填满一个缓冲区时），或者可以调用函数fflush冲洗一个流。值得引起注意的是在UNIX环境中，flush有两种意思。在标准I/O库方面，flush（冲洗）意味着将缓冲区中的内容写到磁盘上（该缓冲区可能只是局部填写的）。在终端驱动程序方面（例如第18章中所述的tcflush函数），flush（刷清）表示丢弃已存储在缓冲区中的数据。

(2) 行缓冲。在这种情况下，当在输入和输出中遇到换行符时，标准I/O库执行I/O操作。这允许我们一次输出一个字符（用标准I/O fputc函数），但只有在写了一行之后才进行实际I/O操作。当流涉及一个终端时（例如标准输入和标准输出），通常使用行缓冲。

对于行缓冲有两个限制。第一，因为标准I/O库用来收集每一行的缓冲区的长度是固定的，所以只要填满了缓冲区，那么即使还没有写一个换行符，也进行I/O操作。第二，任何时候只要通过标准I/O库要求从(a)一个不带缓冲的流，或者(b)一个行缓冲的流（它要求从内核得到数据）得到输入数据，那么就会造成冲洗所有行缓冲输出流。在(b)中带了一个在括号中的说明，其理由是，所需的数据可能已在该缓冲区中，它并不要求在需要数据时才从内核读数据。很明显，从不带缓冲的一个流中进行输入（(a)项）要求当时从内核得到数据。 135

(3) 不带缓冲。标准I/O库不对字符进行缓冲存储。例如，如果用标准I/O函数`fputs`写15个字符到不带缓冲的流中，则该函数很可能用3.8节的`write`系统调用函数将这些字符立即写至相关联的打开文件上。

标准出错流`stderr`通常是不带缓冲的，这就使得出错信息可以尽快显示出来，而不管它们是否含有一个换行符。

ISO C要求下列缓冲特征：

- 当且仅当标准输入和标准输出并不涉及交互式设备时，它们才是全缓冲的。
- 标准出错决不会是全缓冲的。

但是，这并没有告诉我们如果标准输入和输出涉及交互式设备时，它们是不带缓冲的还是行缓冲的；以及标准出错是不带缓冲的还是行缓冲的。很多系统默认使用下列类型的缓冲：

- 标准出错是不带缓冲的。
- 如若是涉及终端设备的其他流，则它们是行缓冲的；否则是全缓冲的。

> 本书讨论的四种平台都遵从标准I/O缓冲的这些惯例，标准出错是不带缓冲的，打开至终端设备的流是行缓冲的；其他所有流则是全缓冲的。

我们将在5.12节和程序清单5-3对标准I/O缓冲作更详细的说明。

对任何一个给定的流，如果我们并不喜欢这些系统默认的情况，则可调用下列两个函数中的一个更改缓冲类型：

```
#include <stdio.h>

void setbuf(FILE *restrict fp, char *restrict buf);

int setvbuf(FILE *restrict fp, char *restrict buf, int mode,
            size_t size);
```

返回值：若成功则返回0，若出错则返回非0值

这些函数一定要在流已被打开后调用（这是十分明显的，因为每个函数都要求一个有效的文件指针作为它的第一个参数），而且也应该在对该流执行任何一个其他操作之前调用。 136

可以使用`setbuf`函数打开或关闭缓冲机制。为了带缓冲进行I/O，参数*buf*必须指向一个长度为`BUFSIZ`的缓冲区（该常量定义在`<stdio.h>`中）。通常在此之后该流就是全缓冲的，但是如果该流与一个终端设备相关，那么某些系统也可将其设置为行缓冲。为了关闭缓冲，将*buf*设置为`NULL`。

使用`setvbuf`，我们可以精确地指定所需的缓冲类型。这是用*mode*参数实现的：

`_IOFBF`　全缓冲
`_IOLBF`　行缓冲
`_IONBF`　不带缓冲

如果指定一个不带缓冲的流，则忽略*buf*和*size*参数。如果指定全缓冲或行缓冲，则*buf*和*size*可选择地指定一个缓冲区及其长度。如果该流是带缓冲的，而*buf*是NULL，则标准I/O库将自动地为该流分配适当长度的缓冲区。适当长度指的是由常量BUFSIZ所指定的值。

> 某些C函数库实现使用stat结构中的成员st_blksize所指定的值（见4.2节）决定最佳I/O缓冲区长度。在本章的后续内容中可以看到，GNU C函数库就使用这种方法。

表5-1列出了这两个函数的动作，以及它们的各个选项。

表5-1　setbuf和setvbuf函数

函数	*mode*	*buf*	缓冲区及长度	缓冲类型
setbuf		非空	长度为BUFSIZ的用户*buf*	全缓冲或行缓冲
		NULL	（无缓冲区）	不带缓冲
setvbuf	_IOFBF	非空	长度为*size*的用户*buf*	全缓冲
		NULL	合适长度的系统缓冲区	
	_IOLBF	非空	长度为*size*的用户*buf*	行缓冲
		NULL	合适长度的系统缓冲区	
	_IONBF	（忽略）	（无缓冲区）	不带缓冲

要了解，如果在一个函数内分配一个自动变量类的标准I/O缓冲区，则从该函数返回之前，必须关闭该流（7.8节将对此作更多讨论）。另外，有些实现将缓冲区的一部分用于存放它自己的管理操作信息，所以可以存放在缓冲区中的实际数据字节数少于*size*。一般而言，应由系统选择缓冲区的长度，并自动分配缓冲区。在这种情况下关闭此流时，标准I/O库将自动释放缓冲区。

任何时候，我们都可强制冲洗一个流。

```
#include <stdio.h>

int fflush(FILE *fp);

                                返回值：若成功则返回0，若出错则返回EOF
```

此函数使该流所有未写的数据都被传送至内核。作为一个特例，如若fp是NULL，则此函数将导致所有输出流被冲洗。

137

5.5　打开流

下列三个函数打开一个标准I/O流。

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int filedes, const char *type);

                   三个函数的返回值：若成功则返回文件指针，若出错则返回NULL
```

这三个函数的区别是：

(1) fopen打开一个指定的文件。

(2) freopen在一个指定的流上打开一个指定的文件，如若该流已经打开，则先关闭该流。若该流已经定向，则freopen清除该定向。此函数一般用于将一个指定的文件打开为一个预定义的流：标准输入、标准输出或标准出错。

(3) fdopen获取一个现有的文件描述符（我们可能从open、dup、dup2、fcntl、pipe、socket、socketpair或accept函数得到此文件描述符），并使一个标准的I/O流与该描述符相结合。此函数常用于由创建管道和网络通信通道函数返回的描述符。因为这些特殊类型的文件不能用标准I/O fopen函数打开，所以我们必须先调用设备专用函数以获得一个文件描述符，然后用fdopen使一个标准I/O流与该描述符相关联。

fopen和freopen是ISO C的所属部分。而ISO C并不涉及文件描述符，所以仅有POSIX.1具有fdopen。

*type*参数指定对该I/O流的读、写方式，ISO C规定*type*参数可以有15种不同的值，它们示于表5-2中。

表5-2 打开标准I/O流的*type*参数

type	说 明
r或rb	为读而打开
w或wb	把文件截短至0长，或为写而创建
a或ab	添加；为在文件尾写而打开，或为写而创建
r+或r+b或rb+	为读和写而打开
w+或w+b或wb+	把文件截短至0长，或为读和写而打开
a+或a+b或ab+	为在文件尾读和写而打开或创建

使用字符b作为*type*的一部分，这使得标准I/O系统可以区分文本文件和二进制文件。因为UNIX内核并不对这两种文件进行区分，所以在UNIX系统环境下指定字符b作为*type*的一部分实际上并无作用。

138

对于fdopen，*type*参数的意义稍有区别。因为该描述符已被打开，所以fdopen为写而打开并不截短该文件。（例如，若该描述符原来是由open函数创建的，而且该文件那时已经存在，则其O_TRUNC标志将决定是否截短该文件。fdopen函数不能截短它为写而打开的任一文件。）另外，标准I/O添写方式也不能用于创建该文件（因为如若一个描述符引用一个文件，则该文件一定已经存在）。

当用添写类型打开一文件后，则每次写都将数据写到文件的当前尾端处。如若有多个进程用标准I/O添写方式打开了同一文件，那么来自每个进程的数据都将正确地写到文件中。

4.4 BSD以前的伯克利版本以及Kernighan和Ritchie[1988] 177页上所示的简单版本中的fopen函数并不能正确地处理添写方式。这些版本在打开流时，调用lseek定位到文件尾端。在涉及多个进程时，为了正确地支持添写方式，该文件必须用O_APPEND标志打开，我们已在3.3节中对此进行了讨论。在每次写前，执行一次lseek操作同样也不能正确工作（如同在3.11节中讨论的一样）。

当以读和写类型打开一文件时（*type*中+符号），具有下列限制：

- 如果中间没有fflush、fseek、fsetpos或rewind，则在输出的后面不能直接跟随输入。
- 如果中间没有fseek、fsetpos或rewind，或者一个输入操作没有到达文件尾端，则

在输入操作之后不能直接跟随输出。

对应于表5-2，我们在表5-3中列出了打开一个流的6种不同的方式。

表5-3　打开一个标准I/O流的6种不同的方式

限　制	r	w	a	r+	w+	a+
文件必须已存在	•			•		
擦除文件以前的内容		•			•	
流可以读	•			•	•	•
流可以写		•	•	•	•	•
流只可在尾端处写			•			•

注意，在指定w或a类型创建一个新文件时，我们无法说明该文件的访问权限位（第3章中所述的open函数和creat函数则能做到这一点）。

除非流引用终端设备，否则按系统默认的情况，流被打开时是全缓冲的。若流引用终端设备，则该流是行缓冲的。一旦打开了流，那么在对该流执行任何操作之前，如果希望，则可使用上一节所述的setbuf和setvbuf改变缓冲的类型。

调用fclose关闭一个打开的流。

139

```
#include <stdio.h>
int fclose(FILE *fp);
```

返回值：若成功则返回0，若出错则返回EOF

在该文件被关闭之前，冲洗缓冲区中的输出数据。丢弃缓冲区中的任何输入数据。如果标准I/O库已经为该流自动分配了一个缓冲区，则释放此缓冲区。

当一个进程正常终止时（直接调用exit函数，或从main函数返回），则所有带未写缓冲数据的标准I/O流都会被冲洗，所有打开的标准I/O流都会被关闭。

5.6　读和写流

一旦打开了流，则可在三种不同类型的非格式化I/O中进行选择，对其进行读、写操作：

(1) 每次一个字符的I/O。一次读或写一个字符，如果流是带缓冲的，则标准I/O函数会处理所有缓冲。

(2) 每次一行的I/O。如果想要一次读或写一行，则使用fgets和fputs。每行都以一个换行符终止。当调用fgets时，应说明能处理的最大行长。5.7节将说明这两个函数。

(3) 直接I/O。fread和fwrite函数支持这种类型的I/O。每次I/O操作读或写某种数量的对象，而每个对象具有指定的长度。这两个函数常用于从二进制文件中每次读或写一个结构。5.9节将说明这两个函数。

> 直接I/O（direct I/O）这个术语来自ISO C标准，有时也被称为二进制I/O、一次一个对象I/O、面向记录的I/O或面向结构的I/O。

（5.11节说明了格式化I/O函数，例如printf和scanf。）

1. 输入函数

以下三个函数可用于一次读一个字符。

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);

三个函数的返回值：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF
```

函数getchar等价于getc(stdin)。前两个函数的区别是getc可被实现为宏，而fgetc则不能实现为宏。这意味着：

(1) getc的参数不应当是具有副作用的表达式。

(2) 因为fgetc一定是一个函数，所以可以得到其地址。这就允许将fgetc的地址作为一个参数传送给另一个函数。

140

(3) 调用fgetc所需时间很可能长于调用getc，因为调用函数通常所需的时间长于调用宏。

这三个函数在返回下一个字符时，会将其unsigned char类型转换为int类型。说明为不带符号的理由是，如果最高位为1也不会使返回值为负。要求整型返回值的理由是，这样就可以返回所有可能的字符值再加上一个已出错或已到达文件尾端的指示值。在<stdio.h>中的常量EOF被要求是一个负值，其值经常是-1。这就意味着不能将这三个函数的返回值存放在一个字符变量中，以后还要将这些函数的返回值与常量EOF相比较。

注意，不管是出错还是到达文件尾端，这三个函数都返回同样的值。为了区分这两种不同的情况，必须调用ferror或feof。

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);

两个函数返回值：若条件为真则返回非0值（真），否则返回0（假）

void clearerr(FILE *fp);
```

在大多数实现中，为每个流在FILE对象中维持了两个标志：

• 出错标志。

• 文件结束标志。

调用clearerr则清除这两个标志。

从流中读取数据以后，可以调用ungetc将字符再压送回流中。

```
#include <stdio.h>

int ungetc(int c, FILE *fp);

返回值：若成功则返回c，若出错则返回EOF
```

压送回到流中的字符以后又可从流中读出，但读出字符的顺序与压送回的顺序相反。应当了解，虽然ISO C允许实现支持任何次数的回送，但是它要求实现提供一次只送回一个字符。我们不能期望一次能送回多个字符。

回送的字符不必一定是上一次读到的字符。不能回送EOF。但是当已经到达文件尾端时，仍可以回送一字符。下次读将返回该字符，再次读则返回EOF。之所以能这样做的原因是一次成功的ungetc调用会清除该流的文件结束标志。

当正在读一个输入流，并进行某种形式的分字或分记号操作时，会经常用到回送字符操作。有时需要先看一看下一个字符，以决定如何处理当前字符。然后就需要方便地将刚查看的字符送回，以便下一次调用getc时返回该字符。如果标准I/O库不提供回送能力，就需将该字符存放到一个我们自己的变量中，并设置一个标志以便判别在下一次需要一个字符时是调用getc，还是从我们自己的变量中取用。

141

> 用ungetc压送回字符时，并没有将它们写到文件中或设备上，只是将它们写回标准I/O库的流缓冲区中。

2. 输出函数

对应于上面所述的每个输入函数都有一个输出函数。

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

三个函数返回值：若成功则返回*c*，若出错则返回EOF

与输入函数一样，putchar(c)等效于putc(c, stdout)，putc可实现为宏，而fputc则不能实现为宏。

5.7　每次一行I/O

下面两个函数提供每次输入一行的功能。

```
#include <stdio.h>
char *fgets(char *restrict buf, int n, FILE *restrict fp);
char *gets(char *buf);
```

两个函数返回值：若成功则返回*buf*，若已到达文件结尾或出错则返回NULL

这两个函数都指定了缓冲区的地址，读入的行将送入其中。gets从标准输入读，而fgets则从指定的流读。

对于fgets，必须指定缓冲区的长度n。此函数一直读到下一个换行符为止，但是不超过$n-1$个字符，读入的字符被送入缓冲区。该缓冲区以null字符结尾。如若该行（包括最后一个换行符）的字符数超过$n-1$，则fgets只返回一个不完整的行，但是，缓冲区总是以null字符结尾。对fgets的下一次调用会继续读该行。

gets是一个不推荐使用的函数。其问题是调用者在使用gets时不能指定缓冲区的长度。这样就可能造成缓冲区溢出（如若该行长于缓冲区长度），写到缓冲区之后的存储空间中，从而产生不可预料的后果。这种缺陷曾被利用，造成1988年的因特网蠕虫事件。有关说明请见1989年6月发行的Communications of the ACM（vol.32, no.6）。gets与fgets的另一个区别是，gets并不将换行符存入缓冲区中。

142

> 这两个函数处理换行符方面的差别与UNIX系统的演进有关。早在V7的手册（1979年）中就说明："为了向后兼容，gets删除换行符，而fgets则保持换行符。"

即使ISO C要求实现提供gets，但请使用fgets，而不要使用gets。

fputs和puts提供每次输出一行的功能。

```
#include <stdio.h>

int fputs(const char *restrict str, FILE *restrict fp);

int puts(const char *str);
```

两个函数返回值：若成功则返回非负值，若出错则返回EOF

函数fputs将一个以null符终止的字符串写到指定的流，尾端的终止符null不写出。注意，这并不一定是每次输出一行，因为它并不要求在null符之前一定是换行符。通常，在null符之前是一个换行符，但并不要求总是如此。

puts将一个以null符终止的字符串写到标准输出，终止符不写出。但是，puts然后又将一个换行符写到标准输出。

puts并不像它所对应的gets那样不安全。但是我们还是应避免使用它，以免需要记住它在最后是否添加了一个换行符。如果总是使用fgets和fputs，那么就会熟知在每行终止处我们必须自己处理换行符。

5.8 标准I/O的效率

使用上一节所述的函数，我们能对标准I/O系统的效率有所了解。程序清单5-1中的程序类似于程序清单3-3中的程序，它使用getc和putc将标准输入复制到标准输出。这两个例程可以实现为宏。

程序清单5-1 用getc和putc将标准输入复制到标准输出

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

143

可以用fgetc和fputc改写该程序，这两个一定是函数，而不是宏（我们没有给出更改源代码的细节）。

最后，我们还编写了一个读、写行的版本，见程序清单5-2。

程序清单5-2 用fgets和fputs将标准输入复制到标准输出

```
#include "apue.h"

int
```

```
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

注意，在程序清单5-1和程序清单5-2的程序中，没有显式地关闭标准I/O流。我们知道exit函数将会冲洗任何未写的数据，然后关闭所有打开的流（我们将在8.5节讨论这一点）。将这三个程序的时间与表3-2中的时间进行比较是很有趣的。表5-4中显示了对同一文件（98.5 MB字节，300万行）进行操作所得的数据。

表5-4 使用标准I/O例程得到的计时结果

函 数	用户CPU（秒）	系统CPU（秒）	时钟时间（秒）	程序正文字节数
表3-2中的最佳时间	0.01	0.18	6.67	
fgets、fputs	2.59	0.19	7.15	139
getc、putc	10.84	0.27	12.07	120
fgetc、fputc	10.44	0.27	11.42	120
表3-2中的单字节时间	124.89	161.65	288.64	

对于这三个标准I/O版本的每一个，其用户CPU时间都大于表3-2中的最佳read版本，因为在每次读一个字符的标准I/O版本中有一个要执行1亿次的循环，而在每次读一行的版本中有一个要执行3 144 984次的循环。在read版本中，其循环只需执行12 611次（对于缓冲区长度为8 192字节）。因为系统CPU时间几乎相同，所以用户CPU时间的差别以及等待I/O结束所消耗时间的差别造成了时钟时间的差别。

系统CPU时间几乎相同，原因是因为所有这些程序对内核提出的读、写请求数基本相同。注意，使用标准I/O例程的一个优点是无需考虑缓冲及最佳I/O长度的选择。在使用fgets时需要考虑最大行长，但是与选择最佳I/O长度比较，这要方便得多。

144

表5-4中的最后一列是每个main函数的文本空间字节数（由C编译产生的机器指令）。从中可见，使用getc和putc的版本与使用fgetc和fputc的版本在文本空间长度方面大体相同。通常，getc和putc实现为宏，但在GNU C库实现中，宏简单地扩展为函数调用。

使用每次一行I/O版本其速度大约是每次一个字符版本的两倍。如果fgets和fputs函数是用getc和putc实现的（例如，见Kernighan和Ritchie[1988]的7.7节），那么，可以预期fgets版本的时间会与getc版本相接近。实际上，每次一行的版本会更慢一些，因为除了现已存在的6百万次函数调用外还需另外增加2亿次函数调用的开销。而在本测试中所用的每次一行函数是用memccpy(3)实现的。通常，为了提高效率，memccpy函数用汇编语言而非C语言编写。正因为如此，每次一行版本才会有较高的速度。

这些时间数字的最后一个有趣的方面在于：fgetc版本较表3-2中BUFFSIZE = 1的版本要快得多。两者都使用了约2亿次的函数调用，而在用户CPU时间方面，fgetc版本的速度大约是

后者的12倍，而在时钟时间方面则稍大于25倍。造成这种差别的原因是：使用read的版本执行了2亿次函数调用，这也就引起2亿次系统调用。而对于fgetc版本，它也执行2亿次函数调用，但是这只引起25 222次系统调用。系统调用与普通的函数调用相比通常需要花费更多的时间。

需要声明的是这些计时结果只在某些系统上才有效。这种计时结果依赖于很多实现的特征，而这种特征对于不同的UNIX系统却可能是不同的。尽管如此，有这样一组数据，并对各种版本的差别作出解释，这有助于我们更好地了解系统。在本节及3.9节中我们了解到的基本事实是，标准I/O库与直接调用read和write函数相比并不慢很多。我们观察到使用getc和putc复制1 MB字节数据大约需0.11秒的CPU时间。对于大多数比较复杂的应用程序，最主要的用户CPU时间是由应用本身的各种处理消耗的，而不是由标准I/O例程消耗的。

5.9 二进制I/O

5.6节和5.7节中的函数以一次一个字符或一次一行的方式进行操作。如果进行二进制I/O操作，那么我们更愿意一次读或写整个结构。如果使用getc或putc读、写一个结构，那么必须循环通过整个结构，每次循环处理一个字节，一次读或写一个字节，这会非常麻烦而且费时。如果使用fputs和fgets，那么因为fputs在遇到null字节时就停止，而在结构中可能含有null字节，所以不能使用它实现读结构的要求。类似地，如果输入数据中包含有null字节或换行符，则fgets也不能正确工作。因此，提供了下列两个函数以执行二进制I/O操作。 145

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj,
             FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
```

两个函数的返回值：读或写的对象数

这些函数有两种常见的用法：

(1) 读或写一个二进制数组。例如，为了将一个浮点数组的第2～5个元素写至一个文件上，可以编写如下程序：

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

其中，指定*size*为每个数组元素的长度，*nobj*为欲写的元素数。

(2) 读或写一个结构。例如，可以编写如下程序：

```
struct {
  short  count;
  long   total;
  char   name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

其中，指定*size*为结构的长度，*nobj*为1（要写的对象数）。

将这两个例子结合起来就可读或写一个结构数组。为了做到这一点，*size*应当是该结构的sizeof，*nobj*应是该数组中的元素个数。

fread和fwrite返回读或写的对象数。对于读，如果出错或到达文件尾端，则此数字可以少于*nobj*。在这种情况下，应调用ferror或feof以判断究竟属于哪一种情况。对于写，如果返回值少于所要求的*nobj*，则出错。

使用二进制I/O的基本问题是，它只能用于读在同一系统上已写的数据。多年之前，这并无问题（那时，所有UNIX系统都运行于PDP-11上），而现在，很多异构系统通过网络相互连接起来，而且，这种情况已经非常普遍。常常有这种情形，在一个系统上写的数据，要在另一个系统上进行处理。在这种环境下，这两个函数可能就不能正常工作，其原因是：

146

(1) 在一个结构中，同一成员的偏移量可能因编译器和系统而异（由于不同的对准要求）。确实，某些编译器有一个选项，选择它的不同值，或者使结构中的各成员紧密包装（这可以节省存储空间，而运行性能则可能有所下降）；或者准确对齐（以便在运行时易于访问结构中的各成员）。这意味着即使在同一个系统上，一个结构的二进制存放方式也可能因编译器选项的不同而不同。

(2) 用来存储多字节整数和浮点值的二进制格式在不同的机器体系结构间也可能不同。

在第16章讨论套接字时，我们将涉及某些相关问题。在不同系统之间交换二进制数据的实际解决方法是使用较高级的协议。关于网络协议使用的交换二进制数据的某些技术，请参阅Rogo[1993]的8.2节或者Stevens、Fenner和Rudoff[2004]的5.18节。

在8.14节中，我们将再回到fread函数，那时将用它读一个二进制结构——UNIX的进程会计记录。

5.10 定位流

有三种方法定位标准I/O流。

(1) ftell和fseek函数。这两个函数自V7以来就存在了，但是它们都假定文件的位置可以存放在一个长整型中。

(2) ftello和fseeko函数。Single UNIX Specification引入了这两个函数，可以使文件偏移量不必一定使用长整型。它们使用off_t数据类型代替了长整型。

(3) fgetpos和fsetpos函数。这两个函数是由ISO C引入的。它们使用一个抽象数据类型fpos_t记录文件的位置。这种数据类型可以定义为记录一个文件位置所需的长度。

需要移植到非UNIX系统上运行的应用程序应当使用fgetpos和fsetpos。

```
#include <stdio.h>

long ftell(FILE *fp);

                    返回值：若成功则返回当前文件位置指示，若出错则返回-1L

int fseek(FILE *fp, long offset, int whence);

                                        返回值：若成功则返回0，若出错则返回非0值

void rewind(FILE *fp);
```

147

对于一个二进制文件，其文件位置指示器是从文件起始位置开始度量，并以字节为计量单位。ftell用于二进制文件时，其返回值就是这种字节位置。为了用fseek定位一个二进制文

件，必须指定一个字节*offset*，以及解释这种偏移量的方式。*whence*的值与3.6节中lseek函数的相同：SEEK_SET表示从文件的起始位置开始，SEEK_CUR表示从当前文件位置开始，SEEK_END表示从文件的尾端开始。ISO C并不要求一个实现对二进制文件支持SEEK_END规范说明，其原因是某些系统要求二进制文件的长度是某个幻数的整数倍，非实际内容部分则充填0。但是在UNIX中，对于二进制文件，SEEK_END是受支持的。

对于文本文件，它们的文件当前位置可能不以简单的字节偏移量来度量。这主要也是在非UNIX系统中，它们可能以不同的格式存放文本文件。为了定位一个文本文件，*whence*一定要是SEEK_SET，而且*offset*只能有两种值：0（绕回到文件的起始位置），或是对该文件调用ftell所返回的值。使用rewind函数也可将一个流设置到文件的起始位置。

除了offset的类型是off_t而非long以外，ftello函数与ftell相同，fseeko函数与fseek相同。

```
#include <stdio.h>

off_t ftello(FILE *fp);

                                返回值：若成功则返回当前文件位置指示，若出错则返回-1

int fseeko(FILE *fp, off_t offset, int whence);

                                返回值：若成功则返回0，若出错则返回非0值
```

回忆3.6节中对off_t数据类型的讨论。实现可将off_t类型定义为长于32位。

正如我们已提及的，fgetpos和fsetpos这两个函数是C标准引进的。

```
#include <stdio.h>

int fgetpos(FILE *restrict fp, fpos_t *restrict pos);

int fsetpos(FILE *fp, const fpos_t *pos);

                                两个函数返回值：若成功则返回0，若出错则返回非0值
```

fgetpos将文件位置指示器的当前值存入由pos指向的对象中。在以后调用fsetpos时，可以使用此值将流重新定位至该位置。 148

5.11 格式化I/O

1. 格式化输出

执行格式化输出处理的是4个printf函数。

```
#include <stdio.h>

int printf(const char *restrict format, ...);

int fprintf(FILE *restrict fp, const char *restrict format, ...);

                                两个函数返回值：若成功则返回输出字符数，若输出出错则返回负值

int sprintf(char *restrict buf, const char *restrict format, ...);

int snprintf(char *restrict buf, size_t n,
             const char *restrict format, ...);

                                两个函数返回值：若成功则返回存入数组的字符数，若编码出错则返回负值
```

printf将格式化数据写到标准输出，fprintf写至指定的流，sprintf将格式化的字符送入数组*buf*中。sprintf在该数组的尾端自动加一个null字节，但该字节不包括在返回值中。

注意，sprintf函数可能会造成由*buf*指向的缓冲区的溢出。调用者有责任确保该缓冲区足够大。为了解决这种缓冲区溢出问题，引入了snprintf函数。在该函数中，缓冲区长度是一个显式参数，超过缓冲区尾端写的任何字符都会被丢弃。如果缓冲区足够大，snprintf函数就会返回写入缓冲区的字符数。与sprintf相同，该返回值不包括结尾的null字节。若snprintf函数返回小于缓冲区长度*n*的正值，那么没有截短输出。若发生了一个编码错误，snprintf则返回负值。

格式说明控制其余参数如何编写，以后又如何显示。每个参数按照转换说明编写，转换说明以字符%开始，除转换说明外，格式字符串中的其他字符将按原样，不经任何修改地被复制输出。一个转换说明有4个可选部分，下面将它们都示于方括号中：

```
%[flags][fldwidth][precision][lenmodifier]convtype
```

表5-5中总结了各种标志（*flages*）。

表5-5　转换说明中的标志部分

标　志	说　明
-	在字段内左对齐输出
+	总是显示带符号转换的符号
（空格）	如果第一个字符不是符号，则在其前面加上一个空格
#	指定另一种转换形式（例如，对于十六进制格式，加0x前缀）
0	添加前导0（而非空格）进行填充

fldwidth说明转换的最小字段宽度。如果转换得到的字符较少，则用空格填充它。字段宽度是一个非负十进制数，或是一个星号（*）。

precision说明整型转换后最少输出数字位数、浮点数转换后小数点后的最少位数、字符串转换后的最大字符数。精度是一个句点（.），后接一个可选的非负十进制整数或一个星号（*）。

149

宽度和精度字段两者皆可为*。此时，一个整型参数指定宽度或精度的值。该整型参数正好位于被转换的参数之前。

lenmodifier说明参数长度。其可能的取值示于表5-6中。

表5-6　转换说明中的长度修饰符

长度修饰符	说　明
hh	有符号或无符号的char
h	有符号或无符号的short
l	有符号或无符号的long或者宽字符
ll	有符号或无符号的long long
j	intmax_t或uintmax_t
z	size_t
t	ptrdiff_t
L	long double

convtype不是可选的。它控制如何解释参数。表5-7中列出了各种转换类型。

表5-7 转换说明中的转换类型部分

转换类型	说　明
d、i	有符号十进制
o	无符号八进制
u	无符号十进制
x、X	无符号十六进制
f、F	double精度浮点数
e、E	指数格式的double精度浮点数
g、G	解释为f、F、e或E，取决于被转换的值
a、A	十六进制指数格式的double精度浮点数
c	字符（若带长度修饰符l，则为宽字符）
s	字符串（若带长度修饰符l，则为宽字符）
p	指向void的指针
n	将到目前为止，所写的字符数写入到指针所指向的无符号整型中
%	%字符
C	宽字符（XSI扩展，等效于lc）
S	宽字符串（XSI扩展，等效于ls）

150

下列4种printf族的变体类似于上面的4种，但是可变参数表（...）代换成了*arg*。

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *restrict format, va_list arg);

int vfprintf(FILE *restrict fp, const char *restrict format,
            va_list arg);

                    两个函数返回值：若成功则返回输出字符数，若输出出错则返回负值

int vsprintf(char *restrict buf, const char *restrict format,
            va_list arg);

int vsnprintf(char *restrict buf, size_t n,
             const char *restrict format, va_list arg);

                    两个函数返回值：若成功则返回存入数组的字符数，若编码出错则返回负值
```

在附录B的出错处理例程中，将使用vsnprintf函数。

关于ISO C标准中有关可变长度参数表的详细说明请参阅Kernighan和Ritchie[1988]的7.3节。应当了解的是，由ISO C提供的可变长度参数表例程（<stdarg.h>头文件和相关的例程）与由较早版本UNIX提供的<varargs.h>例程是不同的。

2. 格式化输入

执行格式化输入处理的是三个scanf函数。

```
#include <stdio.h>

int scanf(const char *restrict format, ...);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

int sscanf(const char *restrict buf, const char *restrict format,
           ...);
```

三个函数返回值：指定的输入项数；若输入出错或在任意变换前已到达文件结尾则返回EOF

scanf族用于分析输入字符串，并将字符序列转换成指定类型的变量。格式之后的各参数包含了变量的地址，以用转换结果初始化这些变量。

151

格式说明控制如何转换参数，以便对它们赋值。转换说明以%字符开始。除转换说明和空白字符外，格式字符串中的其他字符必须与输入匹配。若有一个字符不匹配，则停止后续处理，不再读输入的其余部分。

一个转换说明有三个可选部分，下面将它们都示于方括号中：

```
%[*][fldwidth][lenmodifier]convtype
```

可选的前导星号（*）用于抑制转换。按照转换说明的其余部分对输入进行转换，但转换结果并不存放在参数中。

fldwidth说明最大宽度（即最大字符数）。lenmodifier说明要用转换结果初始化的参数大小。由printf函数族支持的长度修饰符同样得到scanf函数族的支持（见表5-6中的长度修饰符列表）。

convtype字段类似于printf族的转换类型字段，但两者之间还有些差别。一个差别是，存储在无符号类型中的结果可在输入时带上符号。例如，-1可被转换成4 294 967 295赋予无符号整型变量。表5-8列出了scanf函数族支持的转换类型。

表5-8 转换说明中的转换类型

转换类型	说明
d	有符号十进制，基数为10
i	有符号十进制，基数由输入格式决定
o	无符号八进制（输入可选地有符号）
u	无符号十进制，基数为10（输入可选地有符号）
x	无符号十六进制（输入可选地有符号）
a、A、e、E、f、F、g、G	浮点数
c	字符（若带长度修饰符l，则为宽字符）
s	字符串（若带长度修饰符l，则为宽字符串）
[	匹配列出的字符序列，以]终止
[^	匹配除列出字符以外的所有字符，以]终止
p	指向void的指针
n	将到目前为止读取的字符数写入到指针所指向的无符号整型中
%	%字符
C	宽字符（XSI扩展，等效于lc）
S	宽字符串（XSI扩展，等效于ls）

152

与printf族一样，scanf族也支持函数使用由<stdarg.h>说明的可变参数表。

```
#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *restrict format, va_list arg);

int vfscanf(FILE *restrict fp, const char *restrict format,
            va_list arg);

int vsscanf(const char *restrict buf, const char *restrict format,
            va_list arg);
```

三个函数返回值：指定的输入项数，若输入出错或在任一变换前已到达文件结尾则返回EOF

关于scanf函数族的详细情况，请参阅UNIX系统手册。

5.12 实现细节

正如前述，在UNIX系统中，标准I/O库最终都要调用第3章中说明的I/O例程。每个标准I/O流都有一个与其相关联的文件描述符，可以对一个流调用fileno函数以获得其描述符。

> 注意，fileno不是ISO C标准部分，而是POSIX.1支持的扩展。

```
#include <stdio.h>

int fileno(FILE *fp);
```

返回值：与该流相关联的文件描述符

如果要调用dup或fcntl等函数，则需要此函数。

为了了解你所使用的系统中标准I/O库的实现，最好从头文件<stdio.h>开始。从中可以看到：FILE对象是如何定义的、每个流标志的定义以及定义为宏的各个标准I/O例程（例如getc）。Kernighan和Ritchie[1988]中的8.5节含有一个示例实现，从中可以看到很多UNIX实现的基本样式。Plauger[1992]的第12章提供了标准I/O库一种实现的全部源代码。GNU标准I/O库的实现也是公开可以使用的。

实 例

程序清单5-3中的程序为三个标准流以及一个与普通文件相关联的流打印有关缓冲的状态信息。

153

程序清单5-3 对各个标准I/O流打印缓冲状态信息

```
#include "apue.h"

void    pr_stdio(const char *, FILE *);

int
main(void)
{
    FILE    *fp;
```

```
    fputs("enter any character\n", stdout);
    if (getchar() == EOF)
        err_sys("getchar error");
    fputs("one line to standard error\n", stderr);

    pr_stdio("stdin",  stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);

    if ((fp = fopen("/etc/motd", "r")) == NULL)
        err_sys("fopen error");
    if (getc(fp) == EOF)
        err_sys("getc error");
    pr_stdio("/etc/motd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);

    /*
     * The following is nonportable.
     */
    if (fp->_IO_file_flags & _IO_UNBUFFERED)
        printf("unbuffered");
    else if (fp->_IO_file_flags & _IO_LINE_BUF)
        printf("line buffered");
    else /* if neither of above */
        printf("fully buffered");
    printf(", buffer size = %d\n", fp->_IO_buf_end - fp->_IO_buf_base);
}
```

注意，在打印缓冲状态信息之前，先对每个流执行I/O操作，第一个I/O操作通常就造成为该流分配缓冲。结构成员_IO_file_flags、_IO_buf_base、_IO_buf_end和常量_IO_UNBUFFERED、_IO_LINE_BUFFERED是由Linux中的GNU标准I/O库定义的。应当了解，其他UNIX系统可能会有不同的标准I/O库实现。

154

如果运行程序清单5-3中的程序两次，一次使三个标准流与终端相连接，另一次使它们重定向到普通文件，则所得结果是：

```
$ ./a.out                          stdin、stdout和stderr都连至终端
enter any character
                                   键入换行符
one line to standard error
stream = stdin, line buffered, buffer size = 1024
stream = stdout, line buffered, buffer size = 1024
stream = stderr, unbuffered, buffer size = 1
stream = /etc/motd, fully buffered, buffer size = 4096
$ ./a.out < /etc/termcap > std.out 2> std.err
                                   三个流都重定向，再次运行该程序
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 4096
stream = stdout, fully buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/motd, fully buffered, buffer size = 4096
```

从中可见，该系统的默认情况是：当标准输入、输出连至终端时，它们是行缓冲的。行缓冲的长度是1024字节。注意，这并没有将输入、输出的行长限制为1024字节，这只是缓冲区的长度。如果要将2048字节的行写到标准输出，则要进行两次write系统调用。当将这两个流重定向到普通文件时，它们就变成是全缓冲的，其缓冲区长度是该文件系统优先选用的I/O长度（从stat结构中得到的st_blksize值）。从中也可看到，标准出错如它所应该的那样是非缓冲的，而普通文件按系统默认是全缓冲的。 □

5.13 临时文件

ISO C标准I/O库提供了两个函数以帮助创建临时文件。

```
#include <stdio.h>

char *tmpnam(char *ptr);

                                    返回值：指向唯一路径名的指针

FILE *tmpfile(void);

                        返回值：若成功则返回文件指针，若出错则返回NULL
```

tmpnam函数产生一个与现有文件名不同的一个有效路径名字符串。每次调用它时，它都产生一个不同的路径名，最多调用次数是TMP_MAX。TMP_MAX定义在<stdio.h>中。

155

虽然ISO C定义了TMP_MAX，但该标准只要求其值至少应为25。但是，Single UNIX Specification却要求遵循XSI的系统支持其值至少为10 000。虽然此最小值允许一个实现使用4位数字（0000～9999）作为临时文件名，但是，大多数UNIX实现使用的却是大、小写字符。

若*ptr*是NULL，则所产生的路径名存放在一个静态区中，指向该静态区的指针作为函数值返回。下一次再调用tmpnam时，会重写该静态区（这意味着，如果我们调用此函数多次，而且想保存路径名，则我们应当保存该路径名的副本，而不是指针的副本）。如若*ptr*不是NULL，则认为它指向长度至少是L_tmpnam个字符的数组（常量L_tmpnam定义在头文件<stdio.h>中）。所产生的路径名存放在该数组中，*ptr*也作为函数值返回。

tmpfile创建一个临时二进制文件（类型wb+），在关闭该文件或程序结束时将自动删除这种文件。注意，UNIX对二进制文件不作特殊区分。

实 例

程序清单5-4中的程序演示了这两个函数的应用。

程序清单5-4 tmpnam和tmpfile函数实例

```
#include "apue.h"

int
main(void)
{
    char    name[L_tmpnam], line[MAXLINE];
    FILE    *fp;

    printf("%s\n", tmpnam(NULL));       /* first temp name */
```

```
    tmpnam(name);                           /* second temp name */
    printf("%s\n", name);

    if ((fp = tmpfile()) == NULL)           /* create temp file */
        err_sys("tmpfile error");
    fputs("one line of output\n", fp);      /* write to temp file */
    rewind(fp);                             /* then read it back */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("fgets error");
    fputs(line, stdout);                    /* print the line we wrote */

    exit(0);
}
```

执行程序清单5-4中的程序，可得：

```
$ ./a.out
/tmp/fileC1Icwc
/tmp/filemSkHSe
one line of output
```

156

tmpfile函数经常使用的标准UNIX技术是先调用tmpnam产生一个唯一的路径名，然后，用该路径名创建一个文件，并立即unlink它。回忆4.15节，对一个文件解除链接并不会删除其内容，关闭该文件时才删除其内容。而关闭文件可以是显式进行的，也可以在程序终止时删除其内容。

Single UNIX Specification为处理临时文件定义了另外两个函数，它们是XSI的扩展部分。其中第一个是tempnam函数。

```
#include <stdio.h>

char *tempnam(const char *directory, const char *prefix);
```

返回值：指向唯一路径名的指针

tempnam是tmpnam的一个变体，它允许调用者为所产生的路径名指定目录和前缀。对于目录有4种不同的选择，按下列顺序判断其条件是否为真，并且使用第一个为真的作为目录：

(1) 如果定义了环境变量TMPDIR，则用其作为目录。（在7.9节中将说明环境变量。）

(2) 如果参数*directory*非NULL，则用其作为目录。

(3) 将<stdio.h>中的字符串P_tmpdir用作目录。

(4) 将本地目录（通常是/tmp）用作目录。

如果*prefix*非NULL，则它应该是最多包含5个字符的字符串，用其作为文件名的头几个字符。

该函数调用malloc函数分配动态存储区，用其存放所构造的路径名。当不再使用此路径名时就可释放此存储区（7.8节将说明malloc和free函数）。 □

实例

程序清单5-5中的程序显示了tempnam的应用。

程序清单5-5 演示tempnam函数

```
#include "apue.h"

int
main(int argc, char *argv[])
```

```
{
    if (argc != 3)
        err_quit("usage: a.out <directory> <prefix>");

    printf("%s\n", tempnam(argv[1][0] != ' ' ? argv[1] : NULL,
      argv[2][0] != ' ' ? argv[2] : NULL));

    exit(0);
}
```

157

注意，如果命令行参数（目录或前缀）中的任一个以空白开始，则将其作为null指针传送给该函数。下面显示使用该程序的各种方式。

```
$ ./a.out /home/sar TEMP                      指定目录和前缀
/home/sar/TEMPsf00zi
$ ./a.out " " PFX                             使用默认目录：P_tmpdir
/tmp/PFXfBw7Gi
$ TMPDIR=/var/tmp ./a.out /usr/tmp " "        使用环境变量；无前缀
/var/tmp/file8fVYNi                           环境变量覆盖目录
$ TMPDIR=/no/such/dir ./a.out /home/sar/tmp QQQ
/home/sar/tmp/QQQ98s8Ui                       忽略无效环境目录
```

上述选择目录名的四个步骤按序执行，该函数也检查相应的目录名是否有意义。如果该目录并不存在（例如/no/such/dir），则跳过这一步，试探对目录名的下一次选择。从本例中可以看出在本实现中，p_tmpdir目录是/tmp。我们用来设置环境变量的技术（在程序名前指定TMPDIR=）是由Bourne shell、Kornshell和bash使用的。 □

XSI定义的第二个函数是mkstemp。它类似于tmpfile，但是该函数返回的不是文件指针，而是临时文件的打开文件描述符。

```
#include <stdlib.h>

int mkstemp(char *template);
```

返回值：若成功则返回文件描述符，若出错则返回−1

它所返回的文件描述符可用于读、写该文件。临时文件的名字是用*template*字符串参数选择的。该字符串是一个路径名，其最后6个字符设置为XXXXXX。该函数用不同字符代换XXXXXX，以创建唯一路径名。如若mkstemp成功返回，它就会修改*template*字符串以反映临时文件的名字。

与tempfile不同的是，mkstemp创建的临时文件不会自动被删除。如若想从文件系统名字空间中删除该文件，则我们需要自行unlink它。

使用tmpnam和tempnam的一个不足之处是：在返回唯一路径名和应用程序用该路径名创建文件之间有一个时间窗口。在该时间窗口期间，另一个进程可能创建一个同名文件。tempfile和mkstemp函数则不会产生此种问题，可以使用它们代替tmpnam和tempnam。

mktemp函数类似于mkstemp，只不过mktemp只构建一个适用于临时文件的名字，它没有创建一个文件，所以它也有与tmpnam和tempnam相同的不足之处。mktemp函数在Single UNIX Specification中被标记为遗留接口。Single UNIX Specification的未来版本可能将遗留接口全部删除，因此应当避免使用它。

158

5.14 标准I/O的替代软件

标准I/O库并不完善。Korn和Vo[1991]列出了它的很多不足之处，其中，有些属于基本设计，而大多数则与各种不同的实现有关。

标准I/O库的一个不足之处是效率不高，这与它需要复制的数据量有关。当使用每次一行函数fgets和fputs时，通常需要复制两次数据：一次是在内核和标准I/O缓冲之间（当调用read和write时），第二次是在标准I/O缓冲区和用户程序中的行缓冲区之间。快速I/O库[AT&T 1990a中的fio(3)]避免了这一点，其方法是使读一行的函数返回指向该行的指针，而不是将该行复制到另一个缓冲区中。Hume[1988]报告：由于执行了这种更改，grep(1)实用程序的速度增加了两倍。

Korn和Vo[1991]说明了标准I/O库的另一种代替版本：sfio。这一软件包在速度上与fio相近，通常快于标准I/O库。sfio软件包也提供了一些其他标准I/O库所没有的新特征：推广了I/O流，使其不仅可以代表文件，也可代表存储区；可以编写处理模块，并以栈方式将其压入I/O流，这样就可以改变一个流的操作；较好的异常处理等。

Krieger、Stumm和Unrau[1992]说明了另一个替代软件包，它使用了映射文件——mmap函数，我们将在12.9节中说明此函数。该新软件包称为ASI（Alloc Stream Interface）。其编程接口类似于UNIX存储分配函数（malloc、realloc和free，这些将在7.8节中说明）。与sfio软件包相同，ASI使用指针力图减少数据复制量。

许多标准I/O库实现可用于C函数库中，这种C函数库是为内存较小的系统（例如嵌入式系统）设计的。这些实现对于合理内存要求的关注超过对可移植性、速度以及功能性等方面的关注。这类函数库的两种实现是：uClibc C库（见http://www.uclibc.org）和newlibc C库(http://www.source.redhat.com/newlib)。

5.15 小结

大多数UNIX应用程序都使用标准I/O库。本章说明了该库提供的所有函数，以及某些实现细节和效率方面的考虑。应该看到标准I/O库使用了缓冲技术，而它正是产生很多问题，引起许多混淆的一个区域。

习题

5.1 用setvbuf实现setbuf。

5.2 5.8节中的程序利用fgets和fputs函数复制文件，每次I/O操作只复制一行。若将程序中的MAXLINE改为4，当复制的行超过该最大值时会发生什么情况？对此进行解释。

159

5.3 printf返回0值意味着什么？

5.4 下面的代码在一些机器上运行正确，而在另外一些机器运行时出错，解释问题所在。

```
#include    <stdio.h>

int
main(void)
{
    char    c;
```

```
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

5.5 为什么tempnam限制前缀为5个字符？

5.6 对标准I/O流如何使用fsync函数（见3.13节）？

5.7 在程序清单1-5和程序清单1-8的程序中，打印的提示信息没有包含换行符，程序也没有调用fflush函数，请解释输出提示信息的原因是什么？ 160

第6章

系统数据文件和信息

6.1 引言

UNIX系统的正常运行需要使用大量与系统有关的数据文件，例如，口令文件/etc/passwd和组文件/etc/group就是经常由多种程序使用的两个文件。用户每次登录UNIX系统，以及每次执行ls -l命令时都要使用口令文件。

由于历史原因，这些数据文件都是ASCII文本文件，并且使用标准I/O库读这些文件。但是，对于较大的系统，顺序扫描口令文件非常耗时，我们需要能够以非ASCII文本格式存放这些文件，但仍向应用程序提供可以处理任何一种文件格式的接口。针对这些数据文件的可移植接口是本章的主题。本章还介绍了系统标识函数、时间和日期函数。

6.2 口令文件

UNIX系统的口令文件（POSIX.1则将其称为用户数据库）包含了表6-1中所示的各字段，这些字段包含在<pwd.h>中定义的passwd结构中。

注意，POSIX.1只指定了passwd结构包含的10个字段中的5个。大多数平台至少支持其中7个字段。BSD派生的平台支持全部10个字段。

表6-1 /etc/passwd文件中的字段

说　明	struct passwd成员	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
用户名	char *pw_name	•	•	•	•	•
加密口令	char *pw_passwd		•	•	•	•
数值用户ID	uid_t pw_uid	•	•	•	•	•
数值组ID	gid_t pw_gid	•	•	•	•	•
注释字段	char *pw_gecos		•	•	•	•
初始工作目录	char *pw_dir	•	•	•	•	•
初始shell(用户程序)	char *pw_shell	•	•	•	•	•
用户访问类	char *pw_class		•		•	
下次更改口令时间	time_t pw_change		•		•	
账户到期时间	time_t pw_expire		•		•	

由于历史原因，口令文件存储在/etc/passwd中，而且是一个ASCII文件。每一行包含表6-1中所示的各字段，字段之间用冒号分隔。例如，在Linux上，该文件中可能有下列四行：

```
root:x:0:0:root:/root:/bin/bash
```

```
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

关于这些登录项请注意下列各点：

- 通常有一个用户名为root的登录项，其用户ID是0（超级用户）。
- 加密口令字段包含了一个占位字符。在早期的UNIX系统版本中，该字段存放加密口令。将加密口令存放在一个人人可读的文件中构成了一个安全性漏洞，所以现在将加密口令存放在另一个位置。在下一节讨论口令时，我们将详细涉及此问题。
- 口令文件项中的某些字段可能是空。如果加密口令字段为空，这通常就意味着该用户没有口令（不推荐这样做）。squid登录项有一个空白字段：注释字段。空白注释字段不产生任何影响。
- shell字段包含了一个可执行程序名，它被用作该用户的登录shell。若该字段为空，则取系统默认值，通常是/bin/sh。注意，squid登录项的该字段为/dev/null。显然，这是一个设备，不能执行，因此将其用于此处的目的是，阻止任何人以用户squid的名义登录到该系统。

> 很多服务对于帮助它们得以实现的不同守护进程使用不同的用户ID（见第13章），squid项是为实现squid代理缓冲服务而设置的。

162

- 为了阻止一个特定用户登录系统，除使用/dev/null之外，还有若干种替代方法。一种常见的方法是，将/bin/false用作登录shell。它简单地以不成功（非0）状态终止，该shell将此种终止状态判断为假。另一种常见方法是，用/bin/true禁止一个账户。它所做的一切是以成功（0）状态终止。某些系统提供nologin命令，它打印可自定义的出错信息，然后以非0状态终止。
- 使用nobody用户名的目的是，使任何人都可登录至系统，但其用户ID（65534）和组ID（65534）不提供任何特权。该用户ID和组ID只能访问人人皆可读、写的文件（假定用户ID 65534和组ID 65534并不拥有任何文件，而实际情况就应如此）。
- 提供finger(1)命令的某些UNIX系统支持注释字段中的附加信息。其中，各部分之间都用逗号分隔：用户姓名、办公室地点、办公室电话号码以及家庭电话号码等。另外，如果注释字段中的用户姓名是一个&，则将其替换为登录名。例如，可以有下列记录：

```
sar:x:205:105:Steve Rago, SF 5-121, 555-1111, 555-2222:/home/sar:/bin/sh
```

使用finger命令就可打印Steve Rago的有关信息。

```
$ finger -p sar
Login: sar                          Name: Steve Rago
Directory: /home/sar                Shell: /bin/sh
Office: SF 5-121,   555-1111        Home Phone:  555-2222
On since Mon Jan 19 03:57 (EST) on ttyv0 (messages off)
No Mail.
```

即使你所使用的系统并不支持finger命令，这些信息仍可存放在注释字段中，该字段只是一个注释，并不由系统实用程序解释。

某些系统提供了vipw命令，允许管理员使用该命令编辑口令文件。vipw命令串行化对口令文件所做的更改，并且确保所做的更改与其他相关文件保持一致。系统也常常经由图形用户界面（GUI）提供类似的功能。

POSIX.1只定义了两个获取口令文件项的函数。在给出用户登录名或数值用户ID后，这两个函数就能查询相关项。

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

两个函数返回值：若成功则返回指针，若出错则返回NULL

getpwuid函数由ls(1)程序使用，它将i节点中的数值用户ID映射为用户登录名。在键入登录名时，getpwnam函数由login(1)程序使用。

163

这两个函数都返回一个指向passwd结构的指针，该结构已由这两个函数在执行时填入信息。passwd结构通常是相关函数内的静态变量，只要调用相关函数，其内容就会被重写。

如果要查看的只是登录名或用户ID，那么这两个POSIX.1函数能满足要求，但是也有些程序要查看整个口令文件。下列三个函数则可用于此种目的。

```
#include <pwd.h>

struct passwd *getpwent(void);
```

返回值：若成功则返回指针，若出错或到达文件结尾则返回NULL

```
void setpwent(void);

void endpwent(void);
```

基本POSIX.1标准没有定义这三个函数。在Single UNIX Specification中，它们被定义为XSI扩展。因此，预期所有UNIX实现都将提供这些函数。

调用getpwent时，它返回口令文件中的下一个记录项。如同上面所述的两个POSIX.1函数一样，它返回一个由它填写好的password结构的指针。每次调用此函数时都重写该结构。在第一次调用该函数时，它打开它所使用的各个文件。在使用本函数时，对口令文件中各个记录项的安排顺序并无要求。某些系统采用散列算法对/etc/passwd文件中的各项排序。

函数setpwent反绕它所使用的文件，endpwent则关闭这些文件。在使用getpwent查看完口令文件后，一定要调用endpwent关闭这些文件。getpwent知道什么时间它应当打开它所使用的文件（第一次被调用时），但是它并不知道何时关闭这些文件。

实 例

程序清单6-1给出了getpwnam函数的一个实现。

程序清单6-1 getpwnam函数

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
```

```
        if (strcmp(name, ptr->pw_name) == 0)
            break;      /* found a match */
    endpwent();
    return(ptr);    /* ptr is NULL if no match found */
}
```

164

在程序开始处调用setpwent是自我保护性的措施，以便在调用者在此之前已经调用getpwent打开了有关文件情况下，反绕有关文件使它们定位到文件开始处。getpwnam和getpwuid调用完成后不应使有关文件仍处于打开状态，所以应调用endpwent关闭它们。 □

6.3 阴影口令

加密口令是经单向加密算法处理过的用户口令副本。因为此算法是单向的，所以不能从加密口令猜测到原来的口令。

历史上使用的算法（见Morris和Thompson[1979]）总是从64字符集[a-zA-Z0-9./]中产生13个可打印字符。某些较新的系统使用MD5算法对口令加密，为每个加密口令产生31个字符。（加密口令的字符愈多，这些字符的组合也就愈多，于是用各种可能组合来猜测口令的难度就愈大。）当我们将一个字符放在加密口令字段中时，可以确保任一加密口令都不会与其相匹配。

给出一个加密口令，找不到一种算法可以将其逆转到普通文本口令（普通文本口令是在Password:提示符后键入的口令）。但是可以对口令进行猜测，将猜测的口令经单向算法变换成加密形式，然后将其与用户的加密口令相比较。如果用户口令是随机选择的，那么这种方法并不是很有用。但是用户往往以非随机方式选择口令（例如配偶的姓名、街名、宠物名等）。一个经常重复的试验是先得到一份口令文件，然后用试探方法猜测口令。（Garfinkel等[2003]的第4章对UNIX口令及口令加密处理模式的历史情况及细节进行了说明。）

为使企图这样做的人难以获得原始资料（加密口令），现在，某些系统将加密口令存放在另一个通常称为*阴影口令*（shadow password）的文件中。该文件至少要包含用户名和加密口令。与该口令相关的其他信息也可存放在该文件中（表6-2）。

表6-2 etc/shadow文件中的字段

说　明	struct spwd成员
用户登录名	char *sp_namp
加密口令	char *sp_pwdp
上次更改口令以来经过的时间	int sp_lstchg
经过多少天后允许更改	int sp_min
要求更改尚余天数	int sp_max
到期警告天数	int sp_warn
账户不活动之前尚余天数	int sp_inact
账户到期天数	int sp_expire
保留	unsigned int sp_flag

165

只有用户登录名和加密口令这两个字段是必需的。其他字段用于控制口令的改动频率（称为口令的衰老）以及账户保持活动状态的时间。

阴影口令文件不应是一般用户可以读取的。仅有少数几个程序需要存取加密口令，例如login(1)和passwd(1)，这些程序常常是设置用户ID为root的程序。有了阴影口令后，普通

口令文件/etc/passwd可由各用户自由读取。

在Linux 2.4.22和Solaris 9中，与访问口令文件的一组函数类似，有另一组函数可用于访问阴影口令文件。

```
#include <shadow.h>
struct spwd *getspnam(const char *name);
struct spwd *getspent(void);
                        两个函数返回值：若成功则返回指针，若出错则返回NULL
void setspent(void);
void endspent(void);
```

在FreeBSD 5.2.1和MAC OS X 10.3中，没有阴影口令结构。附加的账户信息存放在口令文件中（见表6-1）。

6.4 组文件

UNIX组文件（POSIX.1称其为组数据库）包含了表6-3中所示的字段。这些字段包含在<grp.h>中所定义的group结构中。

表6-3 /etc/group文件中的字段

说 明	struct group成员	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
组名	char *gr_name	•	•	•	•	•
加密口令	char *gr_passwd		•	•	•	•
数值组ID	int gr_gid	•	•	•	•	•
指向各用户名的指针的数组	char **gr_mem	•	•	•	•	•

字段gr_mem是一个指针数组，其中每个指针各指向一个属于该组的用户名。该数组以空指针结尾。

可以用下列两个由POSIX.1定义的函数来查看组名或数值组ID。

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
                        两个函数返回值：若成功则返回指针，若出错则返回NULL
```

166

如同对口令文件进行操作的函数一样，这两个函数通常也返回指向一个静态变量的指针，在每次调用时都重写该静态变量。

如果需要搜索整个组文件，则需使用另外几个函数。下列三个函数类似于针对口令文件的三个函数。

```
#include <grp.h>
struct group *getgrent(void);
                        返回值：若成功则返回指针，若出错或到达文件结尾则返回NULL
void setgrent(void);
void endgrent(void);
```

这三个函数不是基本POSIX.1标准的组成部分。Single UNIX Specification的XSI扩展定义了这些函数。所有UNIX系统都提供这三个函数。

setgrent函数打开组文件（如若它尚未被打开）并反绕它。getgrent函数从组文件中读下一个记录，如若该文件尚未打开则先打开它。endgrent函数关闭组文件。

6.5　附加组ID

在UNIX中，对组的使用已经作了些更改。在V7中，每个用户任何时候都只属于一个组。当用户登录时，系统就按口令文件记录项中的数值组ID，赋给他实际组ID。可以在任何时候执行newgrp(1)以更改组ID。如果newgrp命令执行成功（关于权限规则，请参阅手册），则实际组ID就更改为新的组ID，它将被用于后续的文件访问权限检查。执行不带任何参数的newgrp，则可返回到原来的组。

这种组成员形式一直维持到1983年左右。此时，4.2BSD引入了附加组ID（supplementary group ID）的概念。我们不仅可以属于口令文件记录项中组ID所对应的组，也可属于多达16个另外的组。文件访问权限检查相应被修改为：不仅将进程的有效组ID与文件的组ID相比较，而且也将所有附加组ID与文件的组ID进行比较。

附加组ID是POSIX.1要求的特性。（在较早的POSIX.1版本中，该特性是可选的。）常量NGROUPS_MAX（见表2-10）规定了附加组ID的数量，其常用值是16（见表2-12）。

使用附加组ID的优点是不必再显式地经常更改组。一个用户会参加多个项目，因此也就要同时属于多个组。此类情况是经常有的。

167

为了获取和设置附加组ID，提供了下列三个函数：

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);

                        返回值：若成功则返回附加组ID数，若出错则返回-1

#include <grp.h>    /* on Linux */
#include <unistd.h> /* on FreeBSD, Mac OS X, and Solaris */

int setgroups(int ngroups, const gid_t grouplist[]);

#include <grp.h>    /* on Linux and Solaris */
#include <unistd.h> /* on FreeBSD and Mac OS X */

int initgroups(const char *username, gid_t basegid);

                        两个函数返回值：若成功则返回0，若出错则返回-1
```

在这三个函数中，POSIX.1只说明了getgroups。因为setgroups和initgroups是特权操作，所以它们并非POSIX.1的组成部分。但是，本书说明的所有四种平台都支持这三个函数。

在Mac OS X 10.3中，*basegid*被声明为int类型。

getgroups将各附加组ID填写到数组*grouplist*中，该数组中存放的元素最多为*gidsetsize*个。实际填写到数组中的附加组ID数由函数返回。

作为一个特例，如若*gidsetsize*为0，则函数只返回附加组ID数，而对数组*grouplist*则不作修改（这使调用者可以确定*grouplist*数组的长度，以便进行分配）。

setgroups可由超级用户调用以便为调用进程设置附加组ID表。*grouplist*是组ID数组，而*ngroups*指定了数组中的元素个数。*ngroups*的值不能大于NGROUPS_MAX。

通常，只有initgroups函数调用setgroups，initgroups读整个组文件（用前面说明的函数getgrent、setgrent和endgrent），然后对*username*确定其组的成员关系。然后，它调用setgroups，以便为该用户初始化附加组ID表。因为initgroups调用setgroups，所以只有超级用户才能调用initgroups。除了在组文件中找到*username*是成员的所有组，initgroups也在附加组ID表中包括了*basegid*。*basegid*是*username*在口令文件中的组ID。

只有少数几个程序调用initgroups，例如login(1)程序在用户登录时调用该函数。 168

6.6 实现的区别

我们已讨论了Linux和Solaris支持的阴影口令文件。FreeBSD和MAC OS X则以不同方式存储加密口令。表6-4总结了本书涉及的四种平台如何存放用户和组信息。

表6-4 账户实现的区别

信　息	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
账户信息	/etc/passwd	/etc/passwd	netinfo	/etc/passwd
加密口令	/etc/master.passwd	/etc/shadow	netinfo	/etc/shadow
散列口令文件?	是	否	否	否
组信息	/etc/group	/etc/group	netinfo	/etc/group

在FreeBSD中，阴影口令文件是/etc/master.passwd。可以使用特殊命令编辑该文件，它反过来会从阴影口令文件产生/etc/passwd的一个副本。另外，还会产生该文件的散列版本。/etc/pwd.db是/etc/passwd的散列版本，/etc/spwd.db是/etc/master.passwd的散列版本。这些为大型系统提供了更好的性能。

但是，Mac OS X只以单用户模式使用/etc/passwd和/etc/master.passwd。（在维护系统时，单用户模式通常意味着不能提供任何系统服务。）正常运行期间的多用户方式即netinfo目录服务提供对用户和组账户信息的访问。

虽然Linux和Solaris支持类似的阴影口令接口，但两者之间存在某些微妙的区别。例如，表6-2中所示的整型字段在Solaris中定义为int类型，在Linux中则定义为long int。另一个区别是账户不活动字段。Solaris将其定义为用户上次登录以来所经过的天数，而Linux则将其定义为到口令过期的尚余天数。

在很多系统中，用户和组数据库是用网络信息服务（Network Information Service，NIS）实现的。这使管理员可编辑数据库的主副本，然后将它自动分发到组织中的所有服务器上。客户端系统可以联系服务器以查看用户和组的有关信息。NIS+和轻量级目录访问协议（Lightweight Directory Access Protocol，LDAP）提供了类似功能。很多系统通过配置文件/etc/nsswitch.conf来控制管理每一类信息的方法。

6.7 其他数据文件

至此仅讨论了两个系统数据文件——口令文件和组文件。在日常操作中，UNIX系统还使

169 用很多其他文件。例如，BSD网络软件有一个记录各网络服务器所提供服务的数据文件(/etc/services)，有一个记录协议信息的数据文件（/etc/protocols)，还有一个则是记录网络信息的数据文件（/etc/networks)。幸运的是，针对这些数据文件的接口都与上述针对口令文件和组文件的接口相似。

一般情况下，对于每个数据文件至少有三个函数：

(1) get函数：读下一个记录，如果需要，还可打开该文件。这些函数通常返回指向一个结构的指针。当已到达文件尾端时则返回空指针。大多数get函数返回指向一个静态结构的指针，如果要保存其内容，则需复制它。

(2) set函数：打开相应数据文件（如果尚未打开)，然后反绕该文件。如果希望在相应文件起始处开始处理，则调用此函数。

(3) end函数：关闭相应数据文件。正如前述，在结束了对相应数据文件的读、写操作后，总应调用此函数以关闭所有相关文件。

另外，如果数据文件支持某种形式的关键字搜索，则会提供搜索具有指定关键字记录的例程。例如，对于口令文件，提供了两个按关键字进行搜索的程序：getpwnam寻找具有指定用户名的记录；getpwuid寻找具有指定用户ID的记录。

表6-5中列出了一些这样的例程，这些都是UNIX系统常用的。在表6-5中列出了针对口令文件和组文件的函数，这些已在上面说明过。表6-5中也列出了一些与网络有关的函数。对于表6-5中列出的所有数据文件都有get、set和end函数。

表6-5 存取系统数据文件的类似例程

说明	数据文件	头文件	结 构	附加的关键字查找函数
口令	/etc/passwd	<pwd.h>	passwd	getpwnam、getpwuid
组	/etc/group	<grp.h>	group	getgrnam、getgrgid
阴影	/etc/shadow	<shadow.h>	spwd	getspnam
主机	/etc/hosts	<netdb.h>	hostent	gethostbyname、gethostbyaddr
网络	/etc/networks	<netdb.h>	netent	getnetbyname、getnetbyaddr
协议	/etc/protocols	<netdb.h>	protoent	getprotobyname、getprotobynumber
服务	/etc/services	<netdb.h>	servent	getservbyname、getservbyport

在Solaris中，表6-5中的最后四个数据文件都是符号链接，它们都链接到目录/etc/inet下的同名文件上。大多数UNIX都有类似于该表中所列的附加函数，但是这些附加的函数旨在处理系统管理文件，专用于各个实现。

6.8 登录账户记录

170 大多数UNIX系统都提供下列两个数据文件：utmp文件，它记录当前登录进系统的各个用户；wtmp文件，它跟踪各个登录和注销事件。在V7中，每次写入这两个文件中的是包含下列结构的一条二进制记录：

```
struct utmp {
  char  ut_line[8]; /* tty line: "ttyh0", "ttyd0", "ttyp0", ... */
  char  ut_name[8]; /* login name */
  long  ut_time;    /* seconds since Epoch */
};
```

登录时，login程序填写此类型结构，然后将其写入到utmp文件中，同时也将其添写到wtmp文件中。注销时，init进程将utmp文件中相应的记录擦除（每个字节都填以0），并将一个新记录添写到wtmp文件中。在wtmp文件的注销记录中，将ut_name字段清0。在系统重新启动时，以及更改系统时间和日期的前后，都在wtmp文件中添写特殊的记录项。who(1)程序读utmp文件，并以可读格式打印其内容。后来的UNIX版本提供了last(1)命令，它读wtmp文件并打印所选择的记录。

大多数UNIX版本仍提供utmp和wtmp文件，但正如所期望的，其中的信息量却增加了。V7中20字节的结构在SVR2中已扩充为36字节，而在SVR4中，utmp结构已扩充到多于350字节。

> 在Solaris中，这些记录的详细格式请参见手册页utmpx(4)。在Solaris 9中，这两个文件都在目录/var/adm中。Solaris提供了很多函数（见getutx(3)）用于读或写这两个文件。
>
> 在FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3中，登录记录的格式请参见手册页utmp(5)。这两个文件的路径名是/var/run/utmp和/var/log/wtmp。

6.9 系统标识

POSIX.1定义了uname函数，它返回与当前主机和操作系统有关的信息。

```
#include <sys/utsname.h>

int uname(struct utsname *name);

                                        返回值：若成功则返回非负值，若出错则返回-1
```

通过该函数的参数向其传递一个utsname结构的地址，然后该函数填写此结构。POSIX.1只定义了该结构中至少需提供的字段（它们都是字符数组），而每个数组的长度则由实现确定。某些实现在该结构中提供了另外一些字段。

171

```
struct utsname {
  char  sysname[];    /* name of the operating system */
  char  nodename[];   /* name of this node */
  char  release[];    /* current release of operating system */
  char  version[];    /* current version of this release */
  char  machine[];    /* name of hardware type */
};
```

每个字符串都以null字符结尾。本书讨论的四种平台支持的最大名字长度列于表6-6中。utsname结构中的信息通常可用uname(1)命令打印。

> POSIX.1警告nodename元素可能并不适用于引用通信网络上的主机。此函数来自于系统V，在早期，nodename元素适用于引用UUCP网络上的主机。
>
> 还要认识到在此结构中并没有给出有关POSIX.1版本的信息。应当使用2.6节中所说明的_POSIX_VERSION以获得该信息。
>
> 最后，此函数给出了一种获取该结构中信息的方法，至于如何初始化这些信息，POSIX.1没有给出任何说明。

历史上，BSD派生的系统提供了gethostname函数，它只返回主机名，该名字通常就是TCP/IP网络上主机的名字。

```
#include <unistd.h>

int gethostname(char *name, int namelen);
```

返回值：若成功则返回0，若出错则返回−1

*namelen*参数指定*name*缓冲区长度，如若提供足够的空间，则通过*name*返回的字符串以null结尾。如若没有提供足够的空间，则没有指定通过*name*返回的字符串是否以null结尾。

现在，gethostname函数已定义为POSIX.1的一部分，它指定最大主机名长度是HOST_NAME_MAX。表6-6中列出了本书讨论的四种实现支持的最大名字长度。

表6-6　系统标识名字限制

接　口	最大名字长度			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
uname	256	65	256	257
gethostname	256	64	256	256

172

如果将主机连接到TCP/IP网络中，则此主机名通常是该主机的完全限定域名。

hostname(1)命令可用来获取和设置主机名。（超级用户用一个类似的函数sethostname来设置主机名。）主机名通常在系统自举时设置，它由/etc/rc或init取自一个启动文件。

6.10　时间和日期例程

由UNIX内核提供的基本时间服务是计算自国际标准时间公元1970年1月1日00:00:00以来经过的秒数。1.10节中曾提及这种秒数是以数据类型time_t表示的。我们称它们为日历时间。日历时间包括时间和日期。UNIX在这方面与其他操作系统的区别是：(a) 以国际标准时间而非本地时间计时；(b) 可自动进行转换，例如变换到夏时制；(c) 将时间和日期作为一个量值保存。

time函数返回当前时间和日期。

```
#include <time.h>

time_t time(time_t *calptr);
```

返回值：若成功则返回时间值，若出错则返回−1

时间值总是作为函数值返回。如果参数不为空，则时间值也存放在由*calptr*指向的单元内。

> 在系统V派生的系统中，调用stime(2)函数，在BSD派生的系统中，则调用settimeofday(2)，用于对内核中的当前时间设置初始值。
>
> Single UNIX Specification没有说明系统如何设置其当前时间。

与time函数相比，gettimeofday提供了更高的分辨率（最高为微秒级）。这对某些应用很重要。

```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

返回值：总是返回0

该函数作为XSI扩展定义在Single UNIX Specification中，*tzp*的唯一合法值是NULL，其他值则将产生不确定的结果。某些平台支持用*tzp*说明时区，但这完全依实现而定，并不由Single UNIX Specification定义。

gettimeofday函数将当前时间存放在*tp*指向的timeval结构中，而该结构存储秒和微秒。 173

```
struct timeval {
        time_t tv_sec;    /* seconds */
        long   tv_usec;   /* microseconds */
};
```

一旦取得这种以秒计的整型时间值后，通常要调用另一个时间函数将其转换为人们可读的时间和日期。图6-1说明了各种时间函数之间的关系。(图中以虚线表示的四个函数localtime、mktime、ctime和strftime都受到环境变量TZ的影响。我们将在本节后面对其进行说明。)

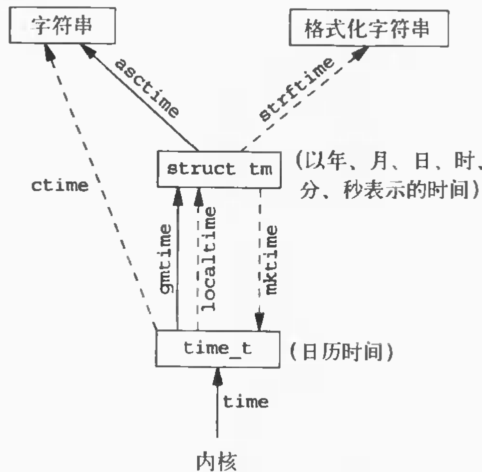


图6-1 各个时间函数之间的关系

两个函数localtime和gmtime将日历时间转换成以年、月、日、时、分、秒、周日表示的时间，并将这些存放在一个tm结构中。

```
struct tm {        /* a broken-down time */
  int   tm_sec;    /* seconds after the minute: [0 - 60] */
  int   tm_min;    /* minutes after the hour: [0 - 59] */
  int   tm_hour;   /* hours after midnight: [0 - 23] */
  int   tm_mday;   /* day of the month: [1 - 31] */
  int   tm_mon;    /* months since January: [0 - 11] */
  int   tm_year;   /* years since 1900 */
  int   tm_wday;   /* days since Sunday: [0 - 6] */
  int   tm_yday;   /* days since January 1: [0 - 365] */
  int   tm_isdst;  /* daylight saving time flag: <0, 0, >0 */
};
```

174

秒可以超过59的理由是可以表示润秒。注意，除了月日字段，其他字段的值都以0开始。如果夏时制生效，则夏时制标志值为正；如果为非夏时制时间，则该标志值为0；如果此信息不可用，则其值为负。

> Single UNIX Specification以前的版本允许双润秒，于是，tm_sec值的有效范围是0~61。UTC的正式定义不允许双润秒，所以，现在tm_sec值的有效范围定义为0~60。

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

两个函数返回值：指向tm结构的指针

localtime和gmtime之间的区别是：localtime将日历时间转换成本地时间（考虑到本地时区和夏时制标志），而gmtime则将日历时间转换成国际标准时间的年、月、日、时、分、秒、周日。

函数mktime以本地时间的年、月、日等作为参数，将其转换成time_t值。

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

返回值：若成功则返回日历时间，若出错则返回-1

asctime和ctime函数产生大家都熟悉的26字节的字符串，这与date(1)命令的系统默认输出形式类似，例如：

```
Tue Feb 10 18:27:38 2004\n\0
```

```
#include <time.h>

char *asctime(const struct tm *tmptr);

char *ctime(const time_t *calptr);
```

两个函数返回值：指向以null结尾的字符串的指针

asctime的参数是指向年、月、日等字符串的指针，而ctime的参数则是指向日历时间的指针。

175

最后一个时间函数是strftime，它是非常复杂的类似于printf的时间值函数。

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr);
```

返回值：若有空间则返回存入数组的字符数，否则返回0

最后一个参数是要格式化的时间值，由一个指向tm结构的指针指定。格式化结果存放在一个长度为*maxsize*个字符的*buf*数组中，如果*buf*长度足以存放格式化结果及一个null终止符，则该函数返回在*buf*中存放的字符数（不包括null终止符），否则该函数返回0。

*format*参数控制时间值的格式。如同printf函数一样，转换说明的形式是百分号之后跟一个特定字符。*format*中的其他字符则按原样输出。两个连续的百分号在输出中产生一个百分号。与printf函数不同的是，每个转换说明产生一个不同的定长输出字符串，在*format*字符串中没有字段宽度修饰符。表6-7中列出了37种ISO C规定的转换说明。表中第三列的数据来自于在Linux中执行strftime函数所得的结果，它对应的时间和日期是：Tue Feb 10 18:27:38 EST 2004。

表6-7 strftime的转换说明

格 式	说 明	实 例
%a	缩写的周日名	Tue
%A	全周日名	Tuesday
%b	缩写的月名	Feb
%B	全月名	February
%c	日期和时间	Tue Feb 10 18:27:38 2004
%C	年/100：[00～99]	20
%d	月日：[01～31]	10
%D	日期[MM/DD/YY]	02/10/04
%e	月日（一位数前加空格）：[1～31]	10
%F	ISO 8601日期格式[YYYY-MM-DD]	2004-02-10
%g	ISO 8601基于周的年的最后2位数[00~99]	04
%G	ISO 8601基于周的年	2004
%h	与%b相同	Feb
%H	小时（24时制）：[00～23]	18
%I	小时（12时制）：[01～12]	06
%j	年日：[001～366]	041
%m	月：[01～12]	02
%M	分：[00～59]	27
%n	换行符	
%p	AM/PM	PM
%r	本地时间：(12时制)	06:27:38 PM
%R	与“%H:%M”相同	18:27
%S	秒：[00～60]	38
%t	水平制表符	
%T	与“%H:%M:%S”相同	18:27:38
%u	ISO 8601周日[Monday=1, 1~7]	2
%U	星期日周数：[00～53]	06
%V	ISO 8601周数: [01～53]	07
%w	周日：[0=Sunday, 06]	2
%W	星期一周数：[00～53]	06
%x	日期	02/10/04
%X	时间	18:27:38
%y	年的最后两位数：[00～99]	04
%Y	年	2004
%z	ISO 8601格式的UTC偏移量	-0500
%Z	时区名	EST
%%	转换为1个%	%

表6-7中的大多数格式说明的意义很明显。需要略作解释的是%U、%V和%W。%U是相应日期在该年中所属周数，包含该年中第一个星期日的周是第一周。%W也是相应日期在该年中所属的周数，不同的是包含第一个星期一的周为第一周。%V说明符则与上述两者有较大区别。若某周包含了1月1日，而且至少包含了其后的另外3天，那么该周是一年中的第一周，否则该周被认为是上一年的最后一周。在这两种情况下，周一都被视作每周的第一天。

如同printf，strftime对某些转换说明支持修饰符。可以使用E和O修饰符产生本地支持的另一种格式。

> 某些系统对strftime的格式字符串提供另外一些非标准的扩充支持。

我们曾在前面提及，图6-1中以虚线表示的四个函数受到环境变量TZ的影响。这四个函数是：localtime、mktime、ctime和strftime。如果定义了TZ，则这些函数将使用其值以代替系统默认时区。如果TZ定义为空串（即TZ=），则使用国际标准时间UTC。TZ的值常常类似于：TZ=EST5EDT，但是POSIX.1允许更详细的说明。有关TZ变量的详细情况，请参阅Single UNIX Specification [Open Group 2004]中的环境变量章节。

> 除gettimeofday函数外，本节所述的其他所有时间和日期函数都是由ISO C标准定义的。在此基础上，POSIX.1增加了环境变量TZ。在FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3中，关于TZ变量的更多信息可参见手册页tzset(3)，在Solaris 9中，可参见手册页environ(5)。

6.11　小结

所有UNIX系统都使用口令文件和组文件。我们说明了读这些文件的各种函数。本章也介绍了阴影口令，它可以增加系统的安全性。附加组ID提供了一个用户同时可以参加多个组的方法。我们还介绍了大多数系统所提供的存取其他与系统有关数据文件的类似函数。我们讨论了几个系统标识函数，应用程序使用它们以标识它在何种系统上运行。最后，说明了ISO C和Single UNIX Specification提供的与时间和日期有关的一些函数。

176～177

习题

6.1　如果系统使用阴影文件，那么如何取得加密口令？

6.2　假设你有超级用户权限，并且系统使用了阴影口令，重新考虑上一道习题。

6.3　编写一个程序，它调用uname并输出utsname结构中的所有字段，将该输出与uname(1)命令的输出结果作比较。

6.4　计算可由time_t数据类型表示的最迟时间。如果超出了这一时间将会如何？

6.5　编写一个程序，获取当前时间，并使用strftime将输出结果转换为类似于date(1)命令的默认输出。将环境变量TZ设置为不同的值，观察输出结果。

178

第7章

进程环境

7.1 引言

下一章将介绍进程控制原语，在此之前需先了解进程的环境。本章中将学习：当执行程序时，其main函数是如何被调用的；命令行参数是如何传送给执行程序的；典型的存储器布局是什么样式；如何分配另外的存储空间；进程如何使用环境变量；各种不同的进程终止方式等。另外，还将说明longjmp和setjmp函数以及它们与栈的交互作用。本章结束之前，还将研究进程的资源限制。

7.2 main函数

C程序总是从main函数开始执行。main函数的原型是

```
int main(int argc, char *argv[]);
```

其中，*argc*是命令行参数的数目，*argv*是指向参数的各个指针所构成的数组。7.4节将对命令行参数进行说明。

当内核执行C程序时（使用一个exec函数，8.10节将说明exec函数），在调用main前先调用一个特殊的启动例程。可执行程序文件将此启动例程指定为程序的起始地址——这是由连接编辑器设置的，而连接编辑器则由C编译器（通常是cc）调用。启动例程从内核取得命令行参数和环境变量值，然后为按上述方式调用main函数做好安排。

179

7.3 进程终止

有8种方式使进程终止（termination），其中5种为正常终止，它们是

(1) 从main返回。

(2) 调用exit。

(3) 调用_exit或_Exit。

(4) 最后一个线程从其启动例程返回（11.5节）。

(5) 最后一个线程调用pthread_exit（11.5节）

异常终止有3种方式，它们是

(6) 调用abort（10.17节）。

(7) 接到一个信号并终止（10.2节）。

(8) 最后一个线程对取消请求做出响应（11.5节和12.7节）。

在第11和12章讨论线程之前，我们暂不考虑专门针对线程的三种终止方式。

上一节提及的启动例程是这样编写的，使得从main返回后立即调用exit函数。如果将启动例程以C代码形式表示（实际上该例程常常用汇编语言编写），则它调用main函数的形式可能是：

```
exit(main(argc, argv));
```

1. exit函数

有三个函数用于正常终止一个程序：_exit和_Exit立即进入内核，exit则先执行一些清理处理（包括调用执行各终止处理程序，关闭所有标准I/O流等），然后进入内核。

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

我们将在8.5节中讨论这三个函数对其他进程（例如正在终止进程的父、子进程）的影响。

180

使用不同头文件的原因是：exit和_Exit是由ISO C说明的，而_exit则是由POSIX.1说明的。

由于历史原因，exit函数总是执行一个标准I/O库的清理关闭操作：为所有打开流调用fclose函数。回忆5.5节，这会造成所有缓冲的输出数据都被冲洗（写到文件上）。

三个exit函数都带一个整型参数，称之为终止状态（或退出状态，exit status）。大多数UNIX shell都提供检查进程终止状态的方法。如果(a)若调用这些函数时不带终止状态，或(b)main执行了一个无返回值的return语句，或(c)main没有声明返回类型为整型，则该进程的终止状态是未定义的。但是，若main的返回类型是整型，并且main执行到最后一条语句时返回（隐式返回），那么该进程的终止状态是0。

这种处理是ISO C标准1999版引入的。历史上，若main函数终止时没有显式使用return语句或调用exit函数，那么进程的终止状态是未定义的。

main函数返回一整型值与用该值调用exit是等价的。于是在main函数中

```
exit(0);
```

等价于

```
return(0);
```

实　例

程序清单7-1中的程序是经典的“hello, world”实例。

程序清单7-1　经典的C程序

```
#include    <stdio.h>

main()
{
    printf("hello, world\n");
}
```

对该程序进行编译，然后运行，则可见到其终止码是随机的。如果在不同的系统上编译该程序，我们很可能得到不同的终止码，这取决于main函数返回时栈和寄存器的内容：

```
$ cc hello.c
$ ./a.out
hello, world
$ echo $?                    打印终止状态
13
```

181

现在，如果我们启用1999 ISO C编译器扩展，则可见到终止码改变了：

```
$ cc -std=c99 hello.c        启用gcc's 1999 ISO C扩展
hello.c:4: warning: return type defaults to `int'
$ ./a.out
hello, world
$ echo $?                    打印终止状态
0
```

注意，当我们启用1999 ISO C扩展时，编译器发出警告消息。打印该警告消息的原因是：main函数的类型没有显式地声明为整型。如果我们增加了这一声明，那么此警告消息就不会出现。但是，如果我们启用编译器所推荐的警告消息（使用-Wall标志），则可能见到类似于“control reaches end of nonvoid function.”控制到达非void函数的尾端这样的警告消息。

将main声明为返回整型，但在main函数体内用exit代替return，对某些C编译器和UNIX lint(1)程序而言会产生不必要的警告信息，因为这些编译器并不知道main中的exit与return语句的作用相同。避开这种警告信息的一种方法是：在main中使用return语句而不是exit。但是这样做的结果是不能用UNIX的grep实用程序来找出程序中所有的exit调用。另外一个解决方案是将main声明为返回void而不是int，然后仍旧调用exit。这也避开了编译器的警告，但从程序设计角度看却并不正确，而且会产生其他的编译警告，因为main的返回类型应当是带符号整型。本章将main表示为返回整型，因为这是ISO C和POSIX.1所指义的。

不同的编译器所发生警告消息的啰嗦程度是不一样的。除非使用警告选项，否则GNU C编译器通常不会发出不必要的警告消息。 □

在下一章，我们将了解到任一进程如何引起一个程序被执行，如何等待进程完成，然后又如何取其终止状态。

2. atexit函数

按照ISO C的规定，一个进程可以登记多达32个函数，这些函数将由exit自动调用。我们称这些函数为终止处理程序（exit handler），并调用atexit函数来登记这些函数。

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

返回值：若成功则返回0，若出错则返回非0值

其中，atexit的参数是一个函数地址，当调用此函数时无需向它传送任何参数，也不期望它返回一个值。exit调用这些函数的顺序与它们登记时候的顺序相反。同一函数如若登记多次，则也会被调用多次。

182

终止处理程序这一机制是由ANSI C标准于1989年引入的。在ANSI C之前出现的系统（例如SVR3和4.3BSD）都没有提供这些终止处理程序。

ISO C要求系统至少应支持32个终止处理程序。为了确定一个给定的平台支持的最大终止处理程

序数，可以使用sysconf函数（见表2-12）。

根据ISO C和POSIX.1，exit首先调用各终止处理程序，然后按需多次调用fclose，关闭所有打开流。POSIX.1扩展了ISO C标准，它指定如若程序调用exec函数族中的任一函数，则将清除所有已安装的终止处理程序。图7-1显示了一个C程序是如何启动的，以及它可以终止的各种方式。

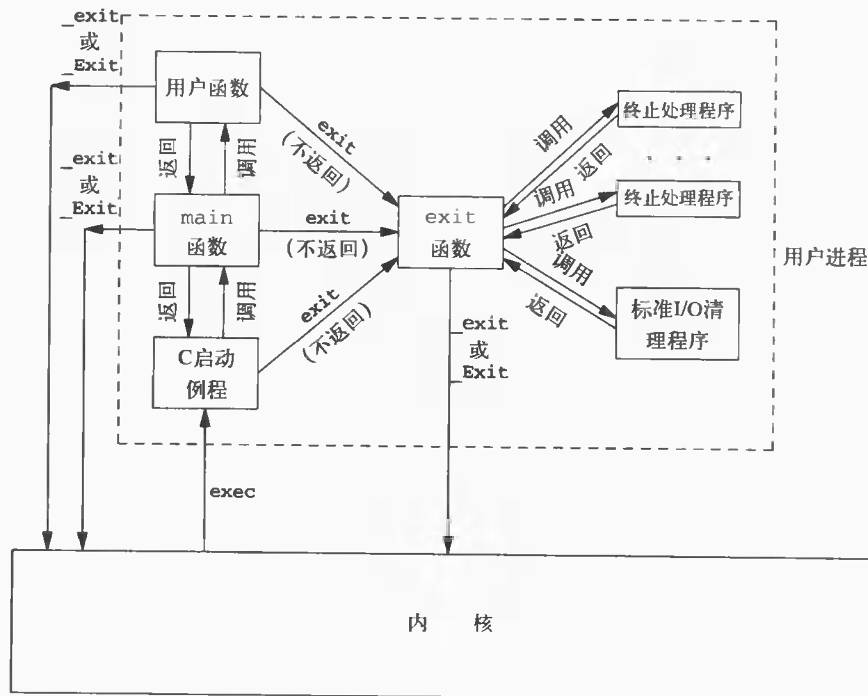


图7-1 一个C程序是如何启动和终止的

注意，内核使程序执行的唯一方法是调用一个exec函数。进程自愿终止的唯一方法是显式或隐式地（通过调用exit）调用_exit或_Exit。进程也可非自愿地由一个信号使其终止（图7-1中没有显示）。

183

实 例

程序清单7-2中的程序说明如何使用atexit函数。

程序清单7-2 终止处理程序实例

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");
```

```
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

执行该程序产生：

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

终止处理程序每登记一次，就会被调用一次。在程序清单7-2中，第一个终止处理程序被登记两次，所以也会被调用两次。注意，在main中没有调用exit，而是用了return语句。 □ 184

7.4 命令行参数

当执行一个程序时，调用exec的进程可将命令行参数传递给该新程序。这是UNIX shell的一部分常规操作。在前几章的很多实例中，我们已经看到了这一点。

实 例

程序清单7-3中的程序将其所有命令行参数都回送到标准输出上。注意，通常的echo（1）程序不会回送第0个参数。

程序清单7-3 将所有命令行参数回送到标准输出

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     i;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

如果编译该程序并将可执行文件命名为echoarg，则得到

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

ISO C和POSIX.1都要求argv[argc]是一个空指针。这就使我们可以将参数处理循环改写为

```
for (i = 0; argv[i] != NULL; i++)
```

□

7.5 环境表

每个程序都会接收到一张环境表。与参数表一样，环境表也是一个字符指针数组，其中每个指针包含一个以null结束的C字符串的地址。全局变量environ则包含了该指针数组的地址：

185

```
extern char **environ;
```

例如，如果该环境包含5个字符串，那么它看起来可能如图7-2所示。其中，每个字符串的结尾处都显式地有一个null字符。我们称environ为环境指针（environment pointer），指针数组为环境表，其中各指针指向的字符串为环境字符串。

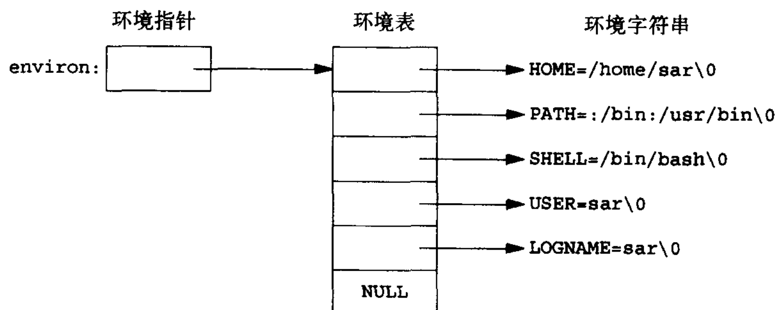


图7-2　由5个C字符串组成的环境

按照惯例，环境由

name=value

这样的字符串组成，这与图7-2中所示相同。大多数预定义的名字完全由大写字母组成，但这只是一个惯例。

在历史上，大多数UNIX系统支持main函数带有三个参数，其中第三个参数就是环境表的地址：

```
int main(int argc, char *argv[], char *envp[]);
```

因为ISO C规定main函数只有两个参数，而且第三个参数与全局变量environ相比也没有带来更多益处，所以POSIX.1也规定应使用environ而不使用第三个参数。通常用getenv和putenv函数（见7.9节）来访问特定的环境变量，而不是用environ变量。但是，如果要查看整个环境，则必须使用environ指针。

7.6 C程序的存储空间布局

从历史上讲，C程序一直由下面几部分组成：

- 正文段。这是由CPU执行的机器指令部分。通常，正文段是可共享的，所以即使是频繁执行的程序（如文本编辑器、C编译器和shell等）在存储器中也只需有一个副本，另外，正文段常常是只读的，以防止程序由于意外而修改其自身的指令。

186

- 初始化数据段。通常将此段称为数据段，它包含了程序中需明确地赋初值的变量。例如，C程序中出现在任何函数之外的声明：

```
int    maxcount = 99;
```

使此变量带有其初值存放在初始化数据段中。

- 非初始化数据段。通常将此段称为bss段，这一名称来源于一个早期的汇编运算符，意思是“block started by symbol”（由符号开始的块），在程序开始执行之前，内核将此段中的数据初始化为0或空指针。出现在任何函数外的C声明

```
long    sum[1000];
```

使此变量存放在非初始化数据段中。

- 栈。自动变量以及每次函数调用时所需保存的信息都存放在此段中。每次调用函数时，其返回地址以及调用者的环境信息（例如某些机器寄存器的值）都存放在栈中。然后，最近被调用的函数在栈上为其自动和临时变量分配存储空间。通过以这种方式使用栈，可以递归调用C函数。递归函数每次调用自身时，就使用一个新的栈帧，因此一个函数调用实例中的变量集不会影响另一个函数调用实例中的变量。
- 堆。通常在堆中进行动态存储分配。由于历史上形成的惯例，堆位于非初始化数据段和栈之间。

图7-3显示了这些段的一种典型安排方式。这是程序的逻辑布局，虽然并不要求一个具体实现一定以这种方式安排其存储空间，但这是一种我们便于说明的典型安排。对于x86处理器上的Linux，正文段从0x08048000单元开始，栈底则在0xC0000000之下开始（在这种特定结构中，栈从高地址向低地址方向增长）。堆顶和栈底之间未用的虚地址空间很大。

187

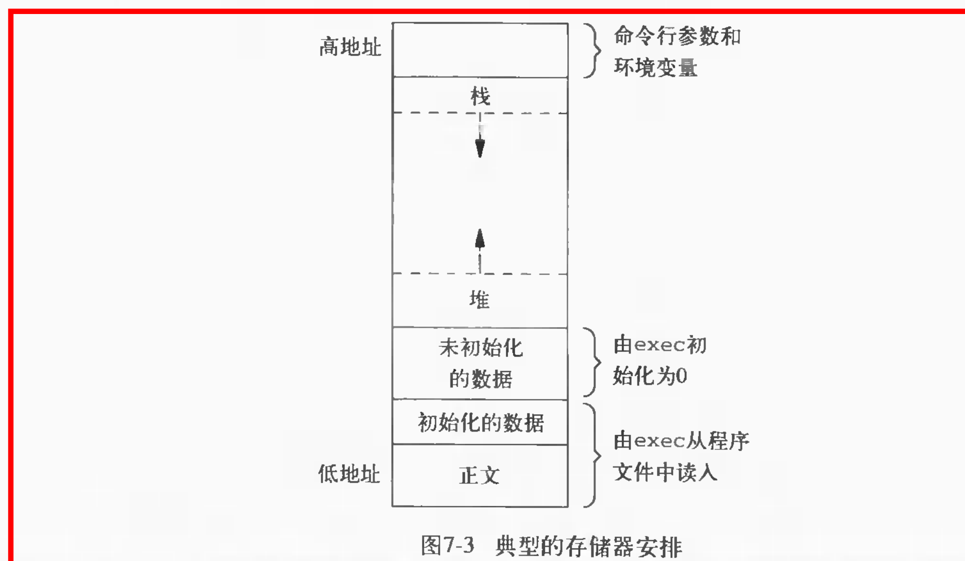


图7-3 典型的存储器安排

a.out中还有若干其他类型的段，例如，包含符号表的段、包含调试信息的段以及包含动态共享库链接表的段等等。这些部分并不装载到进程执行的程序映像中。

从图7-3还可注意到，未初始化数据段的内容并不存放在磁盘上的程序文件中。其原因是，内核在程序开始运行前将它们都设置为0。需要存放在程序文件中的段只有正文段和初始化数据段。

size(1)命令报告正文段、数据段和bss段的长度（单位：字节）。例如：

```
$ size /usr/bin/cc /bin/sh
   text    data    bss     dec     hex   filename
  79606    1536    916   82058   1408a   /usr/bin/cc
 619234   21120  18260  658614   a0cb6   /bin/sh
```

第4列和第5列是分别以十进制和十六进制表示的三个段的总长度。

7.7 共享库

现在，大多数UNIX系统支持共享库。Arnold [1986]说明了系统V上共享库的一个早期实现，Gingell等[1987]则说明了SunOS上的另一个实现。共享库使得可执行文件中不再需要包含公用的库例程，而只需在所有进程都可引用的存储区中维护这种库例程的一个副本。程序第一次执行或者第一次调用某个库函数时，用动态链接方法将程序与共享库函数相链接。这减少了每个可执行文件的长度，但增加了一些运行时间开销。这种时间开销发生在该程序第一次被执行时，或者每个共享库函数第一次被调用时。共享库的另一个优点是可以用库函数的新版本代替老版本，而无需对使用该库的程序重新连接编辑。（假定参数的数目和类型都没有发生改变。）

188

在不同的系统中，程序可能使用不同的方法说明是否要使用共享库。比较典型的有cc(1)和ld(1)命令的选项。作为长度方面发生变化的例子，先用无共享库方式创建下列可执行文件（经典的hello.c程序）：

```
$ cc -static hello1.c          阻止gcc使用共享库
$ ls -l a.out
-rwxrwxr-x  1 sar     475570 Feb 18 23:17 a.out
$ size a.out
   text    data    bss     dec     hex   filename
 375657    3780   3220  382657   5d6c1   a.out
```

如果编译此程序以使用共享库，则可执行文件的正文和数据段的长度都会显著减小：

```
$ cc hello1.c                  gcc默认使用共享库
$ ls -l a.out
-rwxrwxr-x  1 sar      11410 Feb 18 23:19 a.out
$ size a.out
   text    data    bss     dec     hex   filename
    872     256      4    1132     46c   a.out
```

7.8 存储器分配

ISO C说明了三个用于存储空间动态分配的函数。

(1) malloc。分配指定字节数的存储区。此存储区中的初始值不确定。

(2) calloc。为指定数量具指定长度的对象分配存储空间。该空间中的每一位都初始化为0。

(3) realloc。更改以前分配区的长度（增加或减少）。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，以便在尾端提供增加的存储区，而新增区域内的初始值则不确定。

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

                    三个函数返回值：若成功则返回非空指针，若出错则返回NULL

void free(void *ptr);
```

189

这三个分配函数所返回的指针一定是适当对齐的，使其可用于任何数据对象。例如，在一个特定的系统上，如果最苛刻的对齐要求是，double必须在8的倍数地址单元处开始，那么这三个函数返回的指针都应这样对齐。

因为这三个alloc函数都返回通用指针void*，所以如果在程序中包括了#include <stdlib.h>（以获得函数原型），那么当我们将这些函数返回的指针赋予一个不同类型的指针时，就不需要显式地执行类型强制转换。

函数free释放ptr指向的存储空间。被释放的空间通常被送入可用存储区池，以后，可在调用上述三个分配函数时再分配。

realloc函数使我们可以增、减以前分配区的长度（最常见的用法是增加该区）。例如，如果先为一个数组分配存储空间，该数组长度为512，然后在运行时填充它，但运行一段时间后发现该数组原先的长度不够用，此时就可调用realloc扩充相应存储空间。如果在该存储区后有足够的空间可供扩充，则可在原存储区位置上向高地址方向扩充，无需移动任何原先的内容，并返回传送给它的同样的指针值。如果在原存储区后没有足够的空间，则realloc分配另一个足够大的存储区，将现有的512个元素数组的内容复制到新分配的存储区。然后，释放原存储区，返回新分配区的指针。因为这种存储区可能会移动位置，所以不应当使任何指针指到该区中。习题4.16显示了在getcwd中如何使用realloc，以处理任何长度的路径名。程序清单17-28是使用realloc的另一个例子，用其可以避免使用编译时固定长度的数组。

注意，realloc的最后一个参数是存储区的*newsize*（新长度），它不是新、旧存储区长度之差。作为一个特例，若*ptr*是一个空指针，则realloc的功能与malloc相同，用于分配一个指定长度为*newsize*的存储区。

> 这些函数的早期版本允许调用realloc分配自上次malloc、realloc或calloc调用以来所释放的块。这种技巧可回溯到V7，它利用malloc的搜索策略，实现存储器紧缩。Solaris仍支持这一功能，而很多其他平台则不支持。这种功能不被赞同，不应再使用。

这些分配例程通常用sbrk(2)系统调用实现。该系统调用扩充（或缩小）进程的堆（见图7-3）。malloc和free的一个示例实现请见Kernighan和Ritchie [1988]的8.7节。

虽然sbrk可以扩充或缩小进程的存储空间，但是大多数malloc和free的实现都不减小进程的存储空间。释放的空间可供以后再分配，但通常将它们保持在malloc池中而不返回给内核。

应当注意的是，大多数实现所分配的存储空间比所要求的要稍大一些，额外的空间用来记录管理信息——分配块的长度、指向下一个分配块的指针等等。这就意味着如果超过一个已分配区的尾端进行写操作，则会重写后一个块的管理记录。这种类型的错误是灾难性的，但是因为这种错误不会很快暴露出来，所以也就很难发现。同样，在已分配区起始位置之前进行写操作会重写本块的管理记录。

190

在动态分配的缓冲区前或后进行写操作，破坏的可能不仅仅是该区的管理记录信息。在动态分配的缓冲区前后的存储区很可能用于其他动态分配的对象。这些对象与破坏它们的代码可能无关，这造成寻求信息破坏的源头更加困难。

其他可能产生的致命性的错误是：释放一个已经释放了的块；调用free时所用的指针不是三个alloc函数的返回值等。如若一个进程调用malloc函数，但却忘记调用free函数，那么该进程占用的存储器就会连续增加，这被称为泄漏（leakage）。不调用free函数以释放不再使用的空间，那么进程地址空间长度就会慢慢增加，直至不再有空闲空间。此时，由于过度的分页开销，因而使性能下降。

因为存储器分配出错很难跟踪，所以某些系统提供了这些函数的另一种实现版本。每次调用这三个分配函数中的任意一个或free时，它们都进行附加的检错。在调用连接编辑器时指定一个专用库，则在程序中就可使用这种版本的函数。此外还有公共可用的资源，在对其进行编译时使用一个特殊标志就会使附加的运行时检查生效。

FreeBSD、Mac OS X以及Linux通过设置环境变量，支持附加的调试功能。另外，通过符号链接/etc/malloc.conf，可将选项传递给FreeBSD函数库。

1. 替代的存储器分配程序

有很多可替代malloc和free的函数。某些系统已经提供替代存储器分配函数的库。另一些系统只提供标准的存储器分配程序，如果希望，则软件开发者可以下载替代函数。下面讨论某些替代函数和库。

2. libmalloc

基于SVRV的UNIX系统（例如Solaries）包括libmalloc库，它提供了一套与ISO C存储器分配函数相匹配的接口。libmalloc库包括mallopt函数，它使进程可以设置一些变量，并用它们来控制存储空间分配程序的操作。还可使用另一个名为mallinfo的函数，以对存储空间分配程序的操作进行统计。

3. vmalloc

Vo[1996]说明一种存储器分配程序，它允许进程对于不同的存储区使用不同的技术。除了一些vmalloc特有的函数外，该库也提供了ISO C存储器分配的仿真器。

191

4. 快速适配（quick-fit）

历史上，所使用的标准malloc算法是最佳适配或首次适配存储分配策略。快速适配算法快于上述两种算法，但趋向于使用较多的存储器。Weinstock和Wulf[1988]对该算法进行了描述，该算法基于将存储器分裂成各种长度的缓冲区，并将未使用的缓冲区按其长度组成不同的空闲区列表。在许多ftp网站上可方便地取用基于快速适配的malloc和free开放源代码实现。

5. alloca函数

还有一个函数也值得一提，这就是alloca。它的调用序列与malloc相同，但是它是在当前函数的栈帧上分配存储空间，而不是在堆中。其优点是：当函数返回时，自动释放它所使用的栈帧，所以不必再为释放空间而费心。其缺点是：alloca函数增加了栈帧的长度，而某些系统在函数已被调用后不能增加栈帧长度，于是也就不能支持alloca函数。尽管如此，很多软件包还是使用alloca函数，也有很多系统实现了该函数。

本书中讨论的四个平台都提供了alloca函数。

7.9 环境变量

如同前述，环境字符串的形式通常如下：

name=value

UNIX内核并不查看这些字符串，它们的解释完全取决于各个应用程序。例如，shell使用了大量的环境变量。其中某一些在登录时自动设置（如HOME、USER等），有些则由用户设置。我们通常在一个shell启动文件中设置环境变量以控制shell的动作。例如，若设置了环境变量MAILPATH，则它告诉Bourne shell、GNU Bourne-again shell和Korn shell到哪里去查看邮件。

ISO C定义了一个函数getenv，可以用其取环境变量值，但是该标准又称环境的内容是由实现定义的。

```
#include <stdlib.h>

char *getenv(const char *name);
```

返回值：指向与*name*关联的*value*的指针，若未找到则返回NULL

192

注意，此函数返回一个指针，它指向*name* = *value*字符串中的*value*。我们应当使用getenv从环境中取一个指定环境变量的值，而不是直接访问environ。

Single UNIX Specification中的POSIX.1定义了某些环境变量。如果支持XSI扩展，那么其中也包含了另外一些环境变量定义。表7-1列出了由Single UNIX Specification定义的环境变量，并指明本书讨论的4种实现对它们的支持情况。由POSIX.1定义的各环境变量标记为•；否则为XSI扩展。本书讨论的4种UNIX实现使用了很多依赖于实现的环境变量。注意，ISO C没有定义任何环境变量。

表7-1 Single UNIX Specification定义的环境变量

变　量	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说　明
COLUMNS	•	•	•	•	•	终端宽度
DATEMSK	XSI		•		•	getdate(3)模板文件路径名
HOME	•	•	•	•	•	起始目录
LANG	•	•	•	•	•	本地名
LC_ALL	•	•	•	•	•	本地名
LC_COLLATE	•	•	•	•	•	本地排序名
LC_CTYPE	•	•	•	•	•	本地字符分类名
LC_MESSAGES	•	•	•	•	•	本地消息名
LC_MONETARY	•	•	•	•	•	本地货币编辑名
LC_NUMERIC	•	•	•	•	•	本地数字编辑名
LC_TIME	•	•	•	•	•	本地日期/时间格式名
LINES	•	•	•	•	•	终端高度
LOGNAME	•	•	•	•	•	登录名
MSGVERB	XSI	•			•	fmtmsg(3)处理的消息组成部分
NLSPATH	XSI	•	•	•	•	消息类模板序列
PATH	•	•	•	•	•	搜索可执行文件的路径前缀列表
PWD	•	•	•	•	•	当前工作目录的绝对路径名
SHELL	•	•	•	•	•	用户首选的shell名
TERM	•	•	•	•	•	终端类型
TMPDIR	•	•	•	•	•	在其中创建临时文件的目录路径名
TZ	•	•	•	•	•	时区信息

除了取环境变量的值，有时也需要设置环境变量。我们可能希望改变现有变量的值，或者增加新的环境变量。（在下一章将会了解到，我们能影响的只是当前进程及调用的任何子进程的环境，但不能影响父进程的环境，这通常是一个shell进程。尽管如此，修改环境表的能力仍然是很有用的。）不幸的是，并不是所有系统都支持这种能力。表7-2列出了由不同的标准及实现支持的各种函数。

193

表7-2　对于各种环境表函数的支持

函　数	ISO C	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

clearenv不是Single UNIX Specification的组成部分。它被用来删除环境表中的所有项。

表7-2中间三个函数的原型是：

```
#include <stdlib.h>

int putenv(char *str);

int setenv(const char *name, const char *value, int rewrite);

int unsetenv(const char *name);
```

三个函数返回值：若成功则返回0，若出错则返回非0值

这三个函数的操作是：

- putenv取形式为*name* = *value*的字符串，将其放到环境表中。如果*name*已经存在，则先删除其原来的定义。
- setenv将*name*设置为*value*。如果在环境中*name*已经存在，那么(a) 若*rewrite*非0，则首先删除其现有的定义；(b) 若*rewrite*为0，则不删除其现有定义（*name*不设置为新的*value*，而且也不出错）。
- unsetenv删除*name*的定义。即使不存在这种定义也不算出错。

注意putenv和setenv之间的差别。setenv必须分配存储区，以便依据其参数创建*name* = *value*字符串。同时，putenv则无需将传送给它的参数字符串直接放到环境中。确实，在Linux和Solaris中，putenv实现将传送给它的字符串地址作为参数直接放入环境表中。在这种情况下，将存放在栈中的字符串作为参数传送给该函数就会发生错误，其原因是，从当前函数返回时，其栈帧占用的存储区可能将被重用。

这些函数在修改环境表时是如何进行操作的呢？对这一问题进行研究、考察是非常有益的。回忆图7-3，其中，环境表（指向实际*name* = *value*字符串的指针数组）和环境字符串通常存放在进程存储空间的顶部（栈之上）。删除一个字符串很简单——只要先在环境表中找到该指针，然后将所有后续指针都向环境表首部顺次移动一个位置。但是增加一个字符串或修改一个现有的字符串就困难得多。环境表和环境字符串通常占用的是进程地址空间的顶部，所以它不能再

194

向高地址方向（向上）扩展：同时也不能移动在它之下的各栈帧，所以它也不能向低地址方向（向下）扩展。两者组合使得该空间的长度不能再增加。

(1) 如果修改一个现有的*name*:

(a) 如果新*value*的长度少于或等于现有*value*的长度，则只要在原字符串所用空间中写入新字符串。

(b) 如果新*value*的长度大于原长度，则必须调用malloc为新字符串分配空间，然后将新字符串复制到该空间中，接着使环境表中针对*name*的指针指向新分配区。

(2) 如果要增加一个新的*name*，则操作就更加复杂。首先，调用malloc为*name* = *value*字符串分配空间，然后将该字符串复制到此空间中，则：

(a) 如果这是第一次增加一个新*name*，则必须调用malloc为新的指针表分配空间。接着，将原来的环境表复制到新分配区，并将指向新*name* = *value*字符串的指针存放在该指针表的表尾，然后又将一个空指针存放在其后。最后使environ指向新指针表。再看一下图7-3，如果原来的环境表位于栈顶之上（这是一种常见情况），那么必须将此表移至堆中。但是，此表中的大多数指针仍指向栈顶之上的各*name* = *value*字符串。

(b) 如果这不是第一次增加一个新*name*，则可知以前已调用malloc在堆中为环境表分配了空间，所以只要调用realloc，以分配比原空间多存放一个指针的空间。然后将指向新*name* = *value*字符串的指针存放在该表表尾，后面跟着一个空指针。

7.10 setjmp和longjmp函数

在C中，goto语句是不能跨越函数的，而执行这类跳转功能的是函数setjmp和longjmp。这两个函数对于处理发生在深层嵌套函数调用中的出错情况是非常有用的。

考虑程序清单7-4的骨架部分。其主循环是从标准输入读1行，然后调用do_line处理该输入行。do_line函数调用get_token从该输入行中取下一个标记。一行中的第一个标记假定是一条某种形式的命令，于是switch语句就实现命令选择。我们的程序只处理一条命令（add命令），对此命令调用cmd_add函数。

程序清单7-4 进行命令处理的典型程序骨架

```
#include "apue.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */
```

```
void
do_line(char *ptr)       /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
                break;
        }
    }
}

void
cmd_add(void)
{
    int     token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

195
~
196

程序清单7-4中的框架典型地用于读命令，确定命令的类型，然后调用相应函数处理每一条命令这一类程序。图7-4显示了调用了cmd_add之后栈的大致使用情况。

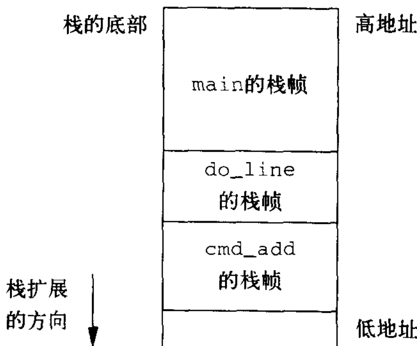


图7-4　调用cmd_add后的各个栈帧

自动变量的存储单元在每个函数的栈帧中。数组line在main的栈帧中，整型cmd在do_line的栈帧中，整型token在cmd_add的栈帧中。

如上所述，这种形式的栈安排是非常典型的，但并不要求非如此不可。栈并不一定要向低地址方向扩充。某些系统对栈并没有提供特殊的硬件支持，此时一个C实现可能要用链接表实现栈帧。

在编写程序清单7-4这样的程序时经常会遇到的一个问题是，如何处理非致命性的错误。例如，若cmd_add函数发现一个错误（譬如说一个无效的数），那么它可能先打印一个出错消息，

然后希望忽略输入行的余下部分，返回main函数并读下一输入行。但是如果这种情况出现在main函数中的深层嵌套中时，用C语言就比较难以做到这一点。（在本例中，cmd_add函数只比main深两个层次，在有些程序中往往深五个层次或更多。）如果我们不得不以检查返回值的方法逐层返回，那就会变得很麻烦。

解决这种问题的方法就是使用非局部goto——setjmp和longjmp函数。非局部指的是，这不是由普通C语言goto语句在一个函数内实施的跳转，而是在栈上跳过若干调用帧，返回到当前函数调用路径上的某一个函数中。

```
#include <setjmp.h>

int setjmp(jmp_buf env);

                    返回值：若直接调用则返回0，若从longjmp调用返回则返回非0值

void longjmp(jmp_buf env, int val);
```

197

在希望返回到的位置调用setjmp，在本例中，此位置在main函数中。因为我们直接调用该函数，所以其返回值为0。setjmp参数*env*的类型是一个特殊类型jmp_buf。这一数据类型是某种形式的数组，其中存放在调用longjmp时能用来恢复栈状态的所有信息。因为需在另一个函数中引用*env*变量，所以规范的处理方式是将*env*变量定义为全局变量。

当检查到一个错误时，例如在cmd_add函数中，则以两个参数调用longjmp函数。第一个就是在调用setjmp时所用的*env*；第二个参数是具有非0值的*val*，它将成为从setjmp处返回的值。使用第二个参数的原因是对于一个setjmp可以有多个longjmp。例如，可以在cmd_add中以*val*为1调用longjmp，也可在get_token中以*val*为2调用longjmp。在main函数中，setjmp的返回值就会是1或2，通过测试返回值就可判断造成返回的longjmp是在cmd_add还是在get_token中。

再回到程序实例中，程序清单7-5给出了经修改过后的main和cmd_add函数（另外两个函数do_line和get_token未经更改）。

程序清单7-5 setjmp和longjmp实例

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

 . . .
```

```
void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)      /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

198

执行main时，调用setjmp，它将所需的信息记入变量jmpbuffer中并返回0。然后调用do_line，它又调用cm_add，假定在其中检测到一个错误。在cmd_add中调用longjmp之前，栈的形式如图7-4中所示。但是longjmp使栈反绕到执行main函数时的情况，也就是抛弃了cmd_add和do_line的栈帧（见图7-5）。调用longjmp造成main中setjmp的返回，但是，这一次的返回值是1（longjmp的第二个参数）。

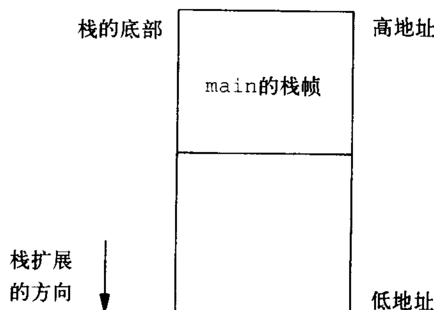


图7-5　调用longjmp后的栈帧

1. 自动、寄存器和易失变量

我们已经了解在调用longjmp后栈帧的基本结构，下一个问题是：在main函数中，自动变量和寄存器变量的状态如何？当longjmp返回到main函数时，这些变量的值是否能恢复到以前调用setjmp时的值（即回滚到原先值），或者这些变量的值保持为调用do_line时的值（do_line调用cmd_add，cmd_add又调用longjmp）？不幸的是，对此问题的回答是“看情况”。大多数实现并不回滚这些自动变量和寄存器变量的值，而所有标准则说它们的值是不确定的。如果你有一个自动变量，而又不想使其值回滚，则可定义其为具有volatile属性。声明为全局或静态变量的值在执行longjmp时保持不变。

下面我们通过程序清单7-6说明在调用longjmp后，自动变量、全局变量、寄存器变量、静态变量和易失变量的不同情况。

199

程序清单7-6　longjmp对各类变量的影响

```
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);
```

```
static jmp_buf  jmpbuffer;
static int      globval;

int
main(void)
{
    int             autoval;
    register int    regival;
    volatile int    volaval;
    static int      statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

200

如果以不带优化和带优化选项对此程序分别进行编译，然后运行它们，则得到的结果是不同的：

```
$ cc testjmp.c                不进行任何优化的编译
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ cc -O testjmp.c             进行全部优化的编译
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

注意，全局、静态和易失变量不受优化的影响，在调用longjmp之后，它们的值是最近所呈现

的值。某个系统的setjmp(3)手册页上说明，存放在存储器中的变量将具有longjmp时的值，而在CPU和浮点寄存器中的变量则恢复为调用setjmp时的值。这确实就是运行程序清单7-6中的程序时所观察到的值。不进行优化时，所有这5个变量都存放在存储器中（亦即忽略了对regival变量的register存储类说明）。而进行了优化后，autoval和regival都存放在寄存器中（即使autoVal并未声明为register），volatile变量则仍存放在存储器中。通过这一实例我们可以理解到，如果要编写一个使用非局部跳转的可移植程序，则必须使用volatile属性。但是从一个系统移植到另一个系统，任何事情都可能改变。

在程序清单7-6中，某些printf的格式字符串可能不适宜安排在程序文本的一行中。我们没有将其分成多个printf调用，而是使用了ISO C的字符串连接功能，于是两个字符串序列

```
"string1" "string2"
```

等价于

```
"string1string2"
```

□

第10章讨论信号处理程序及其信号版本sigsetjmp和siglongjmp时，将再次涉及setjmp和longjmp函数。

2. 自动变量的潜在问题

前面已经说明了处理栈帧的一般方式，现在值得分析一下自动变量的一个潜在出错情况。基本规则是声明自动变量的函数已经返回后，不能再引用这些自动变量。在整个UNIX系统手册中，关于这一点有很多警告。

程序清单7-7中示出了一个名为open_data的函数，它打开了一个标准I/O流，然后为该流设置缓存。

201

程序清单7-7　自动变量的不正确使用

```
#include    <stdio.h>

#define DATAFILE    "datafile"

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];  /* setvbuf makes this the stdio buffer */

    if ((fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp);     /* error */
}
```

问题是：当open_data返回时，它在栈上所使用的空间将由下一个被调用函数的栈帧使用。但是，标准I/O库函数仍将使用其流缓冲区的存储空间。这就产生了冲突和混乱。为了校正这一问题，应在全局存储空间静态地（如static或extern）或者动态地（使用一种alloc函数）为数组databuf分配空间。

7.11 getrlimit和setrlimit函数

每个进程都有一组资源限制，其中一些可以用getrlimit和setrlimit函数查询和更改。

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

两个函数返回值：若成功则返回0，若出错则返回非0值

这两个函数在Single UNIX Specification中定义为XSI扩展。进程的资源限制通常是在系统初始化时由进程0建立的，然后由每个后续进程继承。每种实现都可以用自己的方法对各种限制做出调整。

对这两个函数的每一次调用都会指定一个资源以及一个指向下列结构的指针。

```
struct rlimit {
  rlim_t  rlim_cur;  /* soft limit: current limit */
  rlim_t  rlim_max;  /* hard limit: maximum value for rlim_cur */
};
```

202

在更改资源限制时，须遵循下列三条规则：

(1) 任何一个进程都可将一个软限制值更改为小于或等于其硬限制值。

(2) 任何一个进程都可降低其硬限制值，但它必须大于或等于其软限制值。这种降低对普通用户而言是不可逆的。

(3) 只有超级用户进程可以提高硬限制值。

常量RLIM_INFINITY指定了一个无限量的限制。

这两个函数的*resource*参数取下列值之一。表7-3显示哪些资源限制是由Single UNIX Specification定义并受到本书讨论的4种UNIX系统实现支持的。

表7-3 对资源限制的支持

限 制	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
RLIMIT_AS	•		•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_LOCKS			•		
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_STACK	•	•	•	•	•
RLIMIT_VMEM		•			•

RLIMIT_AS　　进程可用存储区的最大总长度（字节）。这会影响sbrk函数（1.11节）和mmap函数（14.9节）。

RLIMIT_CORE　　core文件的最大字节数，若其值为0则阻止创建core文件。

RLIMIT_CPU　　CPU时间的最大量值（秒），当超过此软限制时，向该进程发送SIGXCPU信号。

RLIMIT_DATA　　数据段的最大字节长度。这是图7-3中初始化数据、非初始以及堆的

	总和。
RLIMIT_FSIZE	可以创建的文件的最大字节长度。当超过此软限制时，则向该进程发送SIGXFSZ信号。
RLIMIT_LOCKS	一个进程可持有的文件锁的最大数（此数也包括Linux特有的文件租借数。详见Linux fcntl(2)手册页）。
RLIMIT_MEMLOCK	一个进程使用mlock(2)能够锁定在存储器中的最大字节长度。
RLIMIT_NOFILE	每个进程能打开的最大文件数。更改此限制将影响到sysconf函数在参数_SC_OPEN_MAX中返回的值（见2.5.4节）。亦见程序清单2-4。
RLIMIT_NPROC	每个实际用户ID可拥有的最大子进程数。更改此限制将影响到sysconf函数在参数_SC_CHILD_MAX中返回的值（见2.5.4节）。
RLIMIT_RSS	最大驻内存集的字节长度（resident set size in bytes，RSS）。如果物理存储器供不应求，则内核将从进程处取回超过RSS的部分。
RLIMIT_SBSIZE	用户在任一给定时刻可以占用的套接字缓冲区的最大长度（字节）。
RLIMIT_STACK	栈的最大字节长度。见图7-3。
RLIMIT_VMEM	这是RLIMIT_AS的同义词。

203

资源限制影响到调用进程并由其子进程继承。这就意味着为了影响一个用户的所有后续进程，需将资源限制的设置构造在shell之中。确实，Bourne shell、GNU Bourne-again shell和Korn shell具有内置的ulimit命令，C shell具有内置的limit命令。（umask和chdir函数也必须是shell内置的。）

实例

程序清单7-8中的程序打印由系统支持的所有资源当前的软限制和硬限制。为了在各种实现上编译该程序，我们已经有条件地包括了各种不同的资源名。另请注意，有些平台定义rlim_t为unsigned long long而非unsigned long，对于此种平台，必须使用不同的printf格式。

程序清单7-8 打印当前资源限制

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld  "
#else
#define FMT "%10ld  "
#endif
#include <sys/resource.h>

#define doit(name)  pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
#ifdef  RLIMIT_AS
    doit(RLIMIT_AS);
#endif
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
```

204

```
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef  RLIMIT_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit   limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
    else
        printf(FMT, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
    else
        printf(FMT, limit.rlim_max);
    putchar((int)'\n');
}
```

205

注意，在doit宏中使用了ISO C的字符串创建运算符（#），以便为每个资源名产生字符串值。例如：

```
doit(RLIMIT_CORE);
```

这将由C预处理程序扩展为：

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

在FreeBSD下运行此程序，得到：

```
$ ./a.out
RLIMIT_CORE     (infinite)  (infinite)
RLIMIT_CPU      (infinite)  (infinite)
RLIMIT_DATA      536870912   536870912
RLIMIT_FSIZE    (infinite)  (infinite)
RLIMIT_MEMLOCK  (infinite)  (infinite)
```

```
RLIMIT_NOFILE          1735        1735
RLIMIT_NPROC            867         867
RLIMIT_RSS       (infinite)  (infinite)
RLIMIT_SBSIZE    (infinite)  (infinite)
RLIMIT_STACK       67108864    67108864
RLIMIT_VMEM      (infinite)  (infinite)
```

在Solaris下运行此程序，得到：

```
$ ./a.out
RLIMIT_AS        (infinite)  (infinite)
RLIMIT_CORE      (infinite)  (infinite)
RLIMIT_CPU       (infinite)  (infinite)
RLIMIT_DATA      (infinite)  (infinite)
RLIMIT_FSIZE     (infinite)  (infinite)
RLIMIT_NOFILE           256       65536
RLIMIT_STACK        8388608  (infinite)
RLIMIT_VMEM      (infinite)  (infinite)
```

□

在介绍了信号机构后，习题10.11将继续讨论资源限制。

7.12 小结

理解UNIX环境中C程序的环境是理解UNIX进程控制特征的先决条件。本章说明了一个进程是如何启动和终止的，如何向其传递参数表和环境。虽然这两者都不是由内核进行解释的，但内核起到了从exec的调用者将这两者传递给新进程的作用。

本章也说明了C程序的典型存储空间布局，以及一个进程如何动态地分配和释放存储器。详细地了解用于维护环境的一些函数是值得的，因为它们涉及存储器分配。本章也介绍了setjmp和longjmp函数，它们提供了一种在进程内执行非局部转移的方法。最后介绍了各种实现提供的资源限制功能。

206

习题

7.1 在Intel x86系统上，无论使用FreeBSD或Linux，如果执行一个输出“hello，world”但不调用exit或return的程序，则程序的返回代码为13（用shell检查），解释其原因。

7.2 程序清单7-2中的printf函数的结果何时才会被真正输出？

7.3 是否有方法不使用（a）参数传递（b）全局变量这两种方法，将main中的参数argc和argv传递给它所调用的其他函数？

7.4 在有些UNIX系统中执行程序时访问不到其数据段的单元0，这是一种有意的安排，为什么？

7.5 用C语言的typdef工具为终止处理程序定义一个新数据类型Exitfunc，使用该类型修改atexit的原型。

7.6 如果用calloc分配一个long型的数组，数组的初始值是否为0？如果用calloc分配一个指针数组，数组的初始值是否为空指针？

7.7 在7.6节size命令的输出结果中，为什么没有给出堆和栈的大小？

7.8 为什么7.7节中两个文件的大小（475570和11410）不等于它们各自的文本和数据大小之和？

7.9 为什么7.7节中对程序使用共享库以后改变了其可执行文件的大小？

7.10 在7.10节末尾，我们已经说明为什么不能将一个指针返回给一个自动变量，下面的程序是否正确？

```
int
f1(int val)
{
    int     *ptr;

    if (val == 0) {
        int     val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```

207

第8章 进程控制

8.1 引言

本章介绍UNIX的进程控制，包括创建新进程、执行程序和进程终止。还将说明进程属性的各种ID——实际、有效和保存的用户和组ID，以及它们如何受到进程控制原语的影响。本章还包括了解释器文件和system函数。本章最后讲述大多数UNIX系统所提供的进程会计机制。这种机制使我们能够从另一个角度了解进程的控制功能。

8.2 进程标识符

每个进程都有一个非负整型表示的唯一进程ID。因为进程ID标识符总是唯一的，常将其用作其他标识符的一部分以保证其唯一性。例如，应用程序有时就把进程ID作为名字的一部分来创建一个唯一的文件名。

虽然是唯一的，但是进程ID可以重用。当一个进程终止后，其进程ID就可以再次使用了。大多数UNIX系统实现延迟重用算法，使得赋予新建进程的ID不同于最近终止进程所使用的ID。这防止了将新进程误认为是使用同一ID的某个已终止的先前进程。

系统中有一些专用的进程，但具体细节因实现而异。ID为0的进程通常是调度进程，常常被称为*交换进程*（swapper）。该进程是内核的一部分，它并不执行任何磁盘上的程序，因此也被称为系统进程。进程ID 1通常是init进程，在自举过程结束时由内核调用。该进程的程序文件在UNIX的早期版本中是/etc/init，在较新版本中是/sbin/init。此进程负责在自举内核后启动一个UNIX系统。init通常读与系统有关的初始化文件（/etc/rc*文件或/etc/inittab文件，以及/etc/init.d中的文件），并将系统引导到一个状态（例如多用户）。init进程决不会终止。它是一个普通的用户进程（与交换进程不同，它不是内核中的系统进程），但是它以超级用户特权运行。本章稍后部分会说明init如何成为所有孤儿进程的父进程。

每个UNIX系统实现都有它自己的一套提供操作系统服务的内核进程，例如，在某些UNIX的虚拟存储器实现中，进程ID 2是页守护进程（pagedaemon）。此进程负责支持虚拟存储系统的分页操作。

除了进程ID，每个进程还有一些其他的标识符。下列函数返回这些标识符。

```
#include <unistd.h>

pid_t getpid(void);
                                        返回值：调用进程的进程ID

pid_t getppid(void);
                                        返回值：调用进程的父进程ID
```

（续）

```
uid_t getuid(void);
```
返回值：调用进程的实际用户ID

```
uid_t geteuid(void);
```
返回值：调用进程的有效用户ID

```
gid_t getgid(void);
```
返回值：调用进程的实际组ID

```
gid_t getegid(void);
```
返回值：调用进程的有效组ID

注意，这些函数都没有出错返回，在下一节中讨论fork函数时，将进一步讨论父进程ID。在4.4节中已讨论了实际和有效用户及组ID。

210

8.3　fork函数

一个现有进程可以调用fork函数创建一个新进程。

```
#include <unistd.h>

pid_t fork(void);
```
返回值：子进程中返回0，父进程中返回子进程ID，出错返回-1

由fork创建的新进程被称为*子进程*（child process）。fork函数被调用一次，但返回两次。两次返回的唯一区别是子进程的返回值是0，而父进程的返回值则是新子进程的进程ID。将子进程ID返回给父进程的理由是：因为一个进程的子进程可以有多个，并且没有一个函数使一个进程可以获得其所有子进程的进程ID。fork使子进程得到返回值0的理由是：一个进程只会有一个父进程，所以子进程总是可以调用getppid以获得其父进程的进程ID（进程ID 0总是由内核交换进程使用，所以一个子进程的进程ID不可能为0）。

子进程和父进程继续执行fork调用之后的指令。子进程是父进程的副本。例如，子进程获得父进程数据空间、堆和栈的副本。注意，这是子进程所拥有的副本。父、子进程并不共享这些存储空间部分。父、子进程共享正文段（见7.6节）。

由于在fork之后经常跟随着exec，所以现在的很多实现并不执行一个父进程数据段、栈和堆的完全复制。作为替代，使用了写时复制（Copy-On-Write, COW）技术。这些区域由父、子进程共享，而且内核将它们的访问权限改变为只读的。如果父、子进程中的任一个试图修改这些区域，则内核只为修改区域的那块内存制作一个副本，通常是虚拟存储器系统中的一"页"。Bach[1986]的9.2节和McKusick等[1996]的5.6节和5.7节对这种特征做了更详细的说明。

> 某些平台提供fork函数的几种变体。本书讨论的四种平台都支持下一节说明的vfork(2)变体。
>
> Linux 2.4.22提供了另一种新进程创建函数——clone(2)系统调用。这是一种fork的泛型，它允许调用者控制哪些部分由父、子进程共享。
>
> FreeBSD 5.2.1提供了rfork(2)系统调用，它类似于Linux的clone系统调用。rfork调用是从Plan 9操作系统（Pike等[1995]）派生出来的。

> Solaris 9提供了两个线程库：一个用于POSIX线程（pthread），另一个用于Solaris线程。在这两个线程库中，fork的特征有所不同。对于POSIX线程，fork创建一个进程，它仅包含调用该fork的线程，但是，对于Solaris线程，fork创建的进程包含了调用线程所在进程的所有线程的副本。为了提供与POSIX线程类似的语义，Solaris提供了fork(1)函数，它创建的进程只复制调用线程，而与所使用的线程库无关。第11和12章将详细讨论线程。

211

实例

程序清单8-1中的程序演示了fork函数，从中可以看到子进程对变量所作的改变并不影响父进程中该变量的值。

程序清单8-1 fork函数示例

```
#include "apue.h"

int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

如果执行此程序则得到：

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89      子进程的变量值改变了
pid = 429, glob = 6, var = 88      父进程的变量值没有改变
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

一般来说，在fork之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用

212 的调度算法。如果要求父、子进程之间相互同步，则要求某种形式的进程间通信。在程序清单8-1中，父进程使自己休眠2秒钟，以使子进程先执行。但并不保证2秒钟已经足够，在8.9节说明竞争条件时，还将谈及这一问题及其他类型的同步方法。在10.16节中，我们将说明在fork之后如何使用信号使父、子进程同步。

当写到标准输出时，我们将buf长度减去1作为输出字节数，这是为了避免将终止null字节写出。strlen计算不包含终止null字节的字符串长度，而sizeof则计算包括终止null字节的缓冲区长度。两者之间的另一个差别是，使用strlen需进行一次函数调用，而对于sizeof而言，因为缓冲区已用已知字符串进行了初始化，其长度是固定的，所以sizeof在编译时计算缓冲区长度。

注意程序清单8-1中fork与I/O函数之间的交互关系。回忆第3章中所述，write函数是不带缓冲的。因为在fork之前调用write，所以其数据写到标准输出一次。但是，标准I/O库是带缓冲的。回忆一下5.12节，如果标准输出连到终端设备，则它是行缓冲的，否则它是全缓冲的。当以交互方式运行该程序时，只得到该printf输出的行一次，其原因是标准输出缓冲区由换行符冲洗。但是当将标准输出重定向到一个文件时，却得到printf输出行两次。其原因是，在fork之前调用了printf一次，但当调用fork时，该行数据仍在缓冲区中，然后在将父进程数据空间复制到子进程中时，该缓冲区也被复制到子进程中。于是那时父、子进程各自有了带该行内容的标准I/O缓冲区。在exit之前的第二个printf将其数据添加到现有的缓冲区中。当每个进程终止时，最终会冲洗其缓冲区中的副本。 □

文件共享

对程序清单8-1需注意的另一点是：在重定向父进程的标准输出时，子进程的标准输出也被重定向。实际上，fork的一个特性是父进程的所有打开文件描述符都被复制到子进程中。父、子进程的每个相同的打开描述符共享一个文件表项（见图3-3）。

考虑下述情况，一个进程具有三个不同的打开文件，它们是标准输入、标准输出和标准出错。在从fork返回时，我们有了如图8-1中所示的结构。

这种共享文件的方式使父、子进程对同一文件使用了一个文件偏移量。考虑下述情况：一个进程fork了一个子进程，然后等待子进程终止。假定，作为普通处理的一部分，父、子进程都向标准输出进行写操作。如果父进程的标准输出已重定向（很可能是由shell实现的），那么子进程写到该标准输出时，它将更新与父进程共享的该文件的偏移量。在我们所考虑的例子中，当父进程等待子进程时，子进程写到标准输出；而在子进程终止后，父进程也写到标准输出上，并且知道其输出会添加在子进程所写数据之后。如果父、子进程不共享同一文件偏移量，213 这种形式的交互就很难实现。

如果父、子进程写到同一描述符文件，但又没有任何形式的同步（例如使父进程等待子进程），那么它们的输出就会相互混合（假定所用的描述符是在fork之前打开的）。虽然这种情况是可能发生的（见图8-1），但这并不是常用的操作模式。

在fork之后处理文件描述符有两种常见的情况：

(1) 父进程等待子进程完成。在这种情况下，父进程无需对其描述符做任何处理。当子进程终止后，它曾进行过读、写操作的任一共享描述符的文件偏移量已执行了相应更新。

(2) 父、子进程各自执行不同的程序段。在这种情况下，在fork之后，父、子进程各自关闭它们不需使用的文件描述符，这样就不会干扰对方使用的文件描述符。这种方法是网络服务进程中经常使用的。

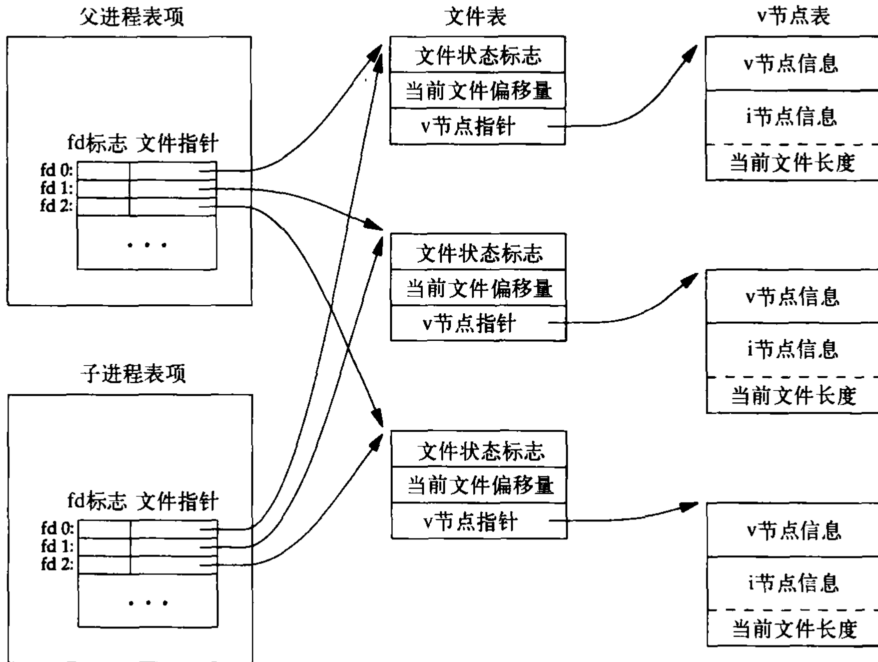


图8-1 调用fork之后父、子进程之间对打开文件的共享

除了打开文件之外，父进程的很多其他属性也由子进程继承，包括

- 实际用户ID、实际组ID、有效用户ID、有效组ID。
- 附加组ID。
- 进程组ID。
- 会话ID。
- 控制终端。
- 设置用户ID标志和设置组ID标志。
- 当前工作目录。
- 根目录。
- 文件模式创建屏蔽字。
- 信号屏蔽和安排。
- 针对任一打开文件描述符的在执行时关闭（close-on-exec）标志。
- 环境。
- 连接的共享存储段。
- 存储映射。
- 资源限制。

214

父、子进程之间的区别是：

- fork的返回值。
- 进程ID不同。
- 两个进程具有不同的父进程ID：子进程的父进程ID是创建它的进程的ID，而父进程的父进程ID则不变。
- 子进程的tms_utime、tms_stime、tms_cutime以及tms_ustime均被设置为0。

- 父进程设置的文件锁不会被子进程继承。
- 子进程的未处理的闹钟（alarm）被清除。
- 子进程的未处理信号集设置为空集。

其中很多特性至今尚未讨论过，我们将在以后几章中对它们进行说明。

使fork失败的两个主要原因是：系统中已经有了太多的进程（通常意味着某个方面出了问题），或者该实际用户ID的进程总数超过了系统限制。回忆表2-10，其中CHILD_MAX规定了每个实际用户ID在任一时刻可具有的最大进程数。

fork有下面两种用法：

(1) 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务进程中是常见的——父进程等待客户端的服务请求。当这种请求到达时，父进程调用fork，使子进程处理此请求。父进程则继续等待下一个服务请求到达。

(2) 一个进程要执行一个不同的程序。这对shell是常见的情况。在这种情况下，子进程从fork返回后立即调用exec（我们将在8.10节说明exec）。

215

某些操作系统将(2)中的两个操作（fork之后执行exec）组合成一个，并称其为spawn。UNIX将这两个操作分开，因为在很多场合需要单独使用fork，其后并不跟随exec。另外，将这两个操作分开，使得子进程在fork和exec之间可以更改自己的属性。例如I/O重定向、用户ID、信号安排等。在第15章中有很多这方面的例子。

Single UNIX Specification在高级实时选项组中确实包括了spawn接口，但是该接口并不打算代替fork和exec。它们的意图是支持难以有效实现fork的系统，特别是对存储管理缺少硬件支持的系统。

8.4 vfork函数

vfork函数的调用序列和返回值与fork相同，但两者的语义不同。

vfork起源于较早的2.9BSD。有些人认为，该函数是有瑕疵的。但是本书讨论的四种平台都支持它。事实上，BSD的开发者在4.4BSD中删除了该函数，但4.4BSD派生的所有开放源码BSD版本又将其收回。在Single UNIX Specification第3版中，vfork被标记为废弃的接口。

vfork用于创建一个新进程，而该新进程的目的是exec一个新程序（与上一节末尾的(2)中一样）。程序清单1-5中的shell基本部分就是这类程序的一个例子。vfork与fork一样都创建一个子进程，但是它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用exec（或exit），于是也就不会存访该地址空间。相反，在子进程调用exec或exit之前，它在父进程的空间中运行。这种优化工作方式在某些UNIX的页式虚拟存储器实现中提高了效率。(这与上一节中提及的在fork之后跟随exec，并采用在写时复制技术相似，而且不复制比部分复制要更快一些。)

vfork和fork之间的另一个区别是：vfork保证子进程先运行，在它调用exec或exit之后父进程才可能被调度运行。(如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。)

实　例

程序清单8-2是程序清单8-1的修改版，其中用vfork代替了fork，删除了对于标准输出的

write调用。另外，我们也不再需要让父进程调用sleep，vfork已保证在子进程调用exec或exit之前，内核会使父进程处于休眠状态。 216

程序清单8-2 vfork函数实例

```
#include "apue.h"

int     glob = 6;       /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

运行该程序得到：

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

子进程对变量glob和var做增1操作，结果改变了父进程中的变量值。因为子进程在父进程的地址空间中运行，所以这并不令人惊讶。但是其作用的确与fork不同。

注意，在程序清单8-2中，调用了_exit而不是exit。正如7.3节所述，_exit并不执行标准I/O缓冲的冲洗操作。如果调用的是exit而不是_exit，则该程序的输出是不确定的。它依赖于标准I/O库的实现，我们可能会见到输出没有发生变化，或者发现没有出现父进程的printf输出。

如果子进程调用exit，而该实现冲洗所有标准I/O流。如果这是函数库采取的唯一动作，那么我们会见到输出与子进程调用_exit所产生的输出完全相同，没有任何区别。如果该实现也关闭标准I/O流，那么表示标准输出FILE对象的相关存储区将被清0。因为子进程借用了父进程的地址空间，所以当父进程恢复运行并调用printf时，也就不会产生任何输出，printf返回-1。注意，父进程的STDOUT_FILENO仍旧有效，子进程得到的是父进程的文件描述符数组的副本（参见图8-1）。 217

大多数exit的现代实现不再在流的关闭方面自找麻烦。因为进程即将终止，那时内核将关闭在进程中已打开的所有文件描述符。在库中关闭它们，只是增加了开销而不会带来任何益处。

□

McKusick等[1996]的5.6节中包含了fork和vfork实现方面的更多信息。习题8.1和8.2则继

续讨论了vfork。

8.5 exit函数

如7.3节所述，进程有下面5种正常终止方式：

(1) 在main函数内执行return语句。如7.3节中所述，这等效于调用exit。

(2) 调用exit函数。此函数由ISO C定义，其操作包括调用各终止处理程序（终止处理程序在调用atexit函数时登记），然后关闭所有标准I/O流等。因为ISO C并不处理文件描述符、多进程（父、子进程）以及作业控制，所以这一定义对UNIX系统而言是不完整的。

(3) 调用_exit或_Exit函数。ISOC定义_Exit，其目的是为进程提供一种无需运行终止处理程序或信号处理程序而终止的方法。对标准I/O流是否进行冲洗，这取决于实现。在UNIX系统中，_Exit和_exit是同义的，并不清洗标准I/O流。_exit函数由exit调用，它处理UNIX特定的细节。_exit是由POSIX.1说明的。

> 在大多数UNIX系统实现中，exit(3)是标准C库中的一个函数，而_exit(2)则是一个系统调用。

(4) 进程的最后一个线程在其启动例程中执行返回语句。但是，该线程的返回值不会用作进程的返回值。当最后一个线程从其启动例程返回时，该进程以终止状态0返回。

(5) 进程的最后一个线程调用pthread_exit函数。如同前面一样，在这种情况中，进程终止状态总是0，这与传送给pthread_exit的参数无关。在11.5节中，我们将对此做更多说明。

三种异常终止方式如下：

(1) 调用abort。它产生SIGABRT信号，这是下一种异常终止的一种特例。

(2) 当进程接收到某些信号时。（第10章将较详细地说明信号。）信号可由进程自身（例如调用abort函数）、其他进程或内核产生。例如，若进程越出其地址空间访问存储单元或者除以0，内核就会为该进程产生相应的信号。

218

(3) 最后一个线程对“取消”（cancellation）请求做出响应。按系统默认，“取消”以延迟方式发生：一个线程要求取消另一个线程，一段时间之后，目标线程终止。在11.5节和11.7节，我们将详细讨论“取消”请求。

不管进程如何终止，最后都会执行内核中的同一段代码。这段代码为相应进程关闭所有打开描述符，释放它所使用的存储器等。

对上述任意一种终止情形，我们都希望终止进程能够通知其父进程它是如何终止的。对于三个终止函数（exit、_exit和_Exit），实现这一点的方法是，将其退出状态（exit status）作为参数传送给函数。在异常终止情况下，内核（不是进程本身）产生一个指示其异常终止原因的终止状态（termination status）。在任意一种情况下，该终止进程的父进程都能用wait或waitpid函数（在下一节说明）取得其终止状态。

注意，这里使用了“退出状态”（它是传向exit或_exit的参数，或main的返回值）和“终止状态”两个术语，以表示有所区别。在最后调用_exit时，内核将退出状态转换成终止状态（回忆图7-1）。下一节中的表8-1说明父进程检查子进程终止状态的不同方法。如果子进程正常终止，则父进程可以获得子进程的退出状态。

在说明fork函数时，显而易见，子进程是在父进程调用fork后生成的。上面又说明了子进程将其终止状态返回给父进程。但是如果父进程在子进程之前终止，则将如何呢？其回答是：对于父进程已经终止的所有进程，它们的父进程都改变为init进程。我们称这些进程由init

进程领养。其操作过程大致如下：在一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止进程的子进程，如果是，则将该进程的父进程ID更改为1（init进程的ID）。这种处理方法保证了每个进程都有一个父进程。

另一个我们关心的情况是如果子进程在父进程之前终止，那么父进程又如何能在做相应检查时得到子进程的终止状态呢？对此问题的回答是：内核为每个终止子进程保存了一定量的信息，所以当终止进程的父进程调用wait或waitpid时，可以得到这些信息。这些信息至少包括进程ID、该进程的终止状态、以及该进程使用的CPU时间总量。内核可以释放终止进程所使用的所有存储区，关闭其所有打开文件。在UNIX术语中，一个已经终止、但是其父进程尚未对其进行善后处理（获取终止子进程的有关信息，释放它仍占用的资源）的进程被称为*僵死进程*（zombie）。ps(1)命令将僵死进程的状态打印为Z。如果编写一个长期运行的程序，它调用fork产生了很多子进程，那么除非父进程等待取得子进程的终止状态，否则这些子进程终止后就会变成僵死进程。

> 某些系统提供了一种避免产生僵死进程的方法，这将在10.7节中介绍。

219

最后一个要考虑的问题是：一个由init进程领养的进程终止时会发生什么？它会不会变成一个僵死进程？对此问题的回答是“否”，因为init被编写成无论何时只要有一个子进程终止，init就会调用一个wait函数取得其终止状态。这样也就防止了在系统中有很多僵死进程。当提及“一个init的子进程”时，这指的可能是init直接产生的进程（例如，将在9.2节说明的getty进程），也可能是其父进程已终止，由init领养的进程。

8.6 wait和waitpid函数

当一个进程正常或异常终止时，内核就向其父进程发送SIGCHLD信号。因为子进程终止是个异步事件（这可以在父进程运行的任何时候发生），所以这种信号也是内核向父进程发的异步通知。父进程可以选择忽略该信号，或者提供一个该信号发生时即被调用执行的函数（信号处理程序）。对于这种信号的系统默认动作是忽略它。第10章将说明这些选项。现在需要知道的是调用wait或waitpid的进程可能会发生什么情况：

- 如果其所有子进程都还在运行，则阻塞。
- 如果一个子进程已终止，正等待父进程获取其终止状态，则取得该子进程的终止状态立即返回。
- 如果它没有任何子进程，则立即出错返回 。

如果进程由于接收到SIGCHLD信号而调用wait，则可期望wait会立即返回。但是如果在任意时刻调用wait，则进程可能会阻塞。

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

两个函数返回值：若成功则返回进程ID，0（见后面的说明），若出错则返回-1

这两个函数的区别如下：

- 在一个子进程终止前，wait使其调用者阻塞，而waitpid有一个选项，可使调用者不阻塞。

- waitpid并不等待在其调用之后的第一个终止子进程，它有若干个选项，可以控制它所等待的进程。

如果一个子进程已经终止，并且是一个僵死进程，则wait立即返回并取得该子进程的状态，否则wait使其调用者阻塞直到一个子进程终止。如调用者阻塞而且它有多个子进程，则在其一个子进程终止时，wait就立即返回。因为wait返回终止子进程的进程ID，所以它总能了解是哪一个子进程终止了。

220

这两个函数的参数*statloc*是一个整型指针。如果*statloc*不是一个空指针，则终止进程的终止状态就存放在它所指向的单元内。如果不关心终止状态，则可将该参数指定为空指针。

依据传统，这两个函数返回的整型状态字是由实现定义的。其中某些位表示退出状态（正常返回），其他位则指示信号编号（异常返回），有一位指示是否产生了一个core文件等。POSIX.1规定终止状态用定义在<sys/wait.h>中的各个宏来查看。有四个互斥的宏可用来取得进程终止的原因，它们的名字都以WIF开始。基于这四个宏中哪一个值为真，就可选用其他宏来取得终止状态、信号编号等。这四个互斥的宏示于表8-1中。

在9.8节中讨论作业控制时，将说明如何停止一个进程。

表8-1　检查wait和waitpid所返回的终止状态的宏

宏	说　明
WIFEXITED (*status*)	若为正常终止子进程返回的状态，则为真。对于这种情况可执行WEXITSTATUS (*status*)，取子进程传送给exit、_exit或_Exit参数的低8位
WIFSIGNALED (*status*)	若为异常终止子进程返回的状态，则为真（接到一个不捕捉的信号）。对于这种情况，可执行WTERMSIG (*status*)，取使子进程终止的信号编号。另外，有些实现（非Single UNIX Specification）定义宏WCOREDUMP (*status*)，若已产生终止进程的core文件，则它返回真
WIFSTOPPED (*status*)	若为当前暂停子进程的返回的状态，则为真。对于这种情况，可执行WSTOPSIG (*status*)，取使子进程暂停的信号编号
WIFCONTINUED (*status*)	若在作业控制暂停后已经继续的子进程返回了状态，则为真。(POSIX.1的XSI扩展，仅用于waitpid。)



程序清单8-3中的函数pr_exit使用表8-1中的宏以打印进程终止状态的说明。本书中的很多程序都将调用此函数。注意，如果定义了WCOREDUMP宏，则此函数也处理该宏。

221

程序清单8-3　打印exit状态的说明

```
#include "apue.h"
#include <sys/wait.h>

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
```

```
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

FreeBSD 5.2.1、Linux 2.4.22、Mac OS X 10.3和Solaris 9都支持WCOREDUMP宏。

程序清单8-4中的程序调用pr_exit函数，演示终止状态的各种值。运行该程序可得

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

不幸的是，没有一种可移植的方法将WTERMSIG得到的信号编号映射为说明性的名字（10.21节中说明了一种方法）。我们必须查看<signal.h>头文件才能知道SIGABRT的值是6，SIGFPE的值是8。 □

程序清单8-4 演示不同的exit值

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;
    int     status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

正如前面所述，如果一个进程有几个子进程，那么只要有一个子进程终止，wait就返回。如果要等待一个指定的进程终止（如果知道要等待进程的ID），那么该如何做呢？在早期的UNIX版本中，必须调用wait，然后将其返回的进程ID和所期望的进程ID相比较。如果终止进程不是所期望的，则将该进程ID和终止状态保存起来，然后再次调用wait。反复这样做直到所期望的进程终止。下一次又想等待一个特定进程时，先查看已终止的进程列表，若其中已有要等待的进程，则取有关信息，否则调用wait。其实，我们需要的是等待一个特定进程的函数。POSIX.1定义了waitpid函数以提供这种功能（以及其他一些功能）。

对于waitpid函数中*pid*参数的作用解释如下：

pid == −1　　等待任一子进程。就这一方面而言，waitpid与wait等效。

pid > 0　　等待其进程ID与*pid*相等的子进程。

222～223

pid == 0　　等待其组ID等于调用进程组ID的任一子进程。（9.4节将说明进程组。）

pid < −1　　等待其组ID等于*pid*绝对值的任一子进程。

waitpid函数返回终止子进程的进程ID，并将该子进程的终止状态存放在由*statloc*指向的存储单元中。对于wait，其唯一的出错是调用进程没有子进程（函数调用被一个信号中断时，也可能返回另一种出错。第10章将对此进行讨论）。但是对于waitpid，如果指定的进程或进程组不存在，或者参数*pid*指定的进程不是调用进程的子进程则都将出错。

*options*参数使我们能进一步控制waitpid的操作。此参数可以是0，或者是表8-2中常量按位“或”运算的结果。

表8-2　waitpid的*options*常量

常　量	说　明
WCONTINUED	若实现支持作业控制，那么由*pid*指定的任一子进程在暂停后已经继续，但其状态尚未报告，则返回其状态（POSIX.1的XSI扩展）
WNOHANG	若由*pid*指定的子进程并不是立即可用的，则waitpid不阻塞，此时其返回值为0
WUNTRACED	若某实现支持作业控制，而由*pid*指定的任一子进程已处于暂停状态，并且其状态自暂停以来还未报告过，则返回其状态。WIFSTOPPED宏确定返回值是否对应于一个暂停子进程

Solaris支持另一个但非标准的选项常量WNOWAIT，它使系统将终止状态已由waitpid返回的进程保持在等待状态，这样它可被再次等待。

waitpid函数提供了wait函数没有提供的三个功能：

(1) waitpid可等待一个特定的进程，而wait则返回任一终止子进程的状态。在讨论popen函数时会再说明这一功能。

(2) waitpid提供了一个wait的非阻塞版本。有时用户希望取得一个子进程的状态，但不想阻塞。

(3) waitpid支持作业控制（利用WUNTRACED和WCONTINUED选项）。

实　例

回忆8.5节中有关僵死进程的讨论。如果一个进程fork一个子进程，但不要它等待子进程终止，也不希望子进程处于僵死状态直到父进程终止，实现这一要求的技巧是调用fork两次。程序清单8-5实现了这一点。

224

程序清单8-5 调用fork两次以避免僵死进程

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

第二个子进程调用sleep以保证在打印父进程ID时第一个子进程已终止。在fork之后，父、子进程都可继续执行，并且我们无法预知哪一个会先执行。在fork之后，如果不使第二个子进程休眠，那么它可能比其父进程先执行，于是它打印的父进程ID将是创建它的父进程，而不是init进程（进程ID 1）。

执行程序清单8-5中的程序得到：

```
$ ./a.out
$ second child, parent pid = 1
```

注意，当原先的进程（也就是exec本程序的进程）终止时，shell打印其提示符，这在第二个子进程打印其父进程ID之前。 □ 225

8.7 waitid函数

Single UNIX Specification的XSI扩展包括了另一个取进程终止状态的函数——waitid，此函数类似于waitpid，但提供了更多的灵活性。

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

返回值：若成功则返回0，若出错则返回−1

与waitpid相似，waitid允许一个进程指定要等待的子进程。但它使用单独的参数表示要等待的子进程的类型，而不是将此与进程ID或进程组ID组合成一个参数。*id*参数的作用与*idtype*的值相关。该函数支持的*idtype*类型列出在表8-3中。

表8-3 waitid的*idtype*常量

常量	说明
P_PID	等待一个特定的进程：*id*包含要等待子进程的进程ID
P_PGID	等待一个特定进程组中的任一子进程：*id*包含要等待子进程的进程组ID
P_ALL	等待任一子进程：忽略*id*

*options*参数是表8-4中各标志的按位"或"。这些标志指示调用者关注哪些状态变化。

表8-4 waitid的*options*常量

常量	说明
WCONTINUED	等待一个进程，它以前曾被暂停，此后又已继续，但其状态尚未报告
WEXITED	等待已退出的进程
WNOHANG	如无可用的子进程退出状态，立即返回而非阻塞
WNOWAIT	不破坏子进程退出状态。该子进程退出状态可由后续的wait、waitid或waitpid调用取得
WSTOPPED	等待一个进程，它已经暂停，但其状态尚未报告

infop参数是指向siginfo结构的指针。该结构包含了有关引起子进程状态改变的生成信号的详细信息。10.14节将进一步讨论siginfo结构。

226

本书讨论的四种平台中只有Solaris支持waitid。

8.8 wait3和wait4函数

大多数UNIX系统实现提供了另外两个函数wait3和wait4。历史上，这两个函数是从UNIX系统的BSD分支沿袭下来的。它们提供的功能比POSIX.1函数wait、waitpid和waitid所提供的功能要多一个，这与附加参数*rusage*有关。该参数要求内核返回由终止进程及其所有子进程使用的资源汇总。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

两个函数返回值：若成功则返回进程ID，若出错则返回−1

资源统计信息包括用户CPU时间总量、系统CPU时间总量、页面出错次数、接收到信号的

次数等。有关细节请参阅getrusage(2)手册页（这种资源信息与7.11节中所述的资源限制不同）。表8-5列出了各个wait函数所支持的参数。

表8-5 不同系统上各个wait函数所支持的参数

函 数	*pid*	*options*	*rusage*	POSIX.1	Free BSD 5.2.1	Linux 2.4.22	Mac OSX 10.3	Solaris 9
wait				•	•	•	•	•
waitid	•	•		XSI				•
waitpid	•	•		•	•	•	•	•
wait3		•	•		•	•	•	•
wait4	•	•	•		•	•	•	•

Single UNIX Specification的早期版本包括wait3函数。在其版本2中，wait3被移到了遗留目录下，在其版本3中，则删去了wait3。

8.9 竞争条件

从本书的目的出发，当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件（race condition）。如果在fork之后的某种逻辑显式或隐式地依赖于在fork之后是父进程先运行还是子进程先运行，那么fork函数就会是竞争条件活跃的滋生地。通常，我们不能预料哪一个进程先运行。即使知道哪一个进程先运行，那么在该进程开始运行后，所发生的事情也依赖于系统负载以及内核的调度算法。

227

在程序清单8-5中，当第二个子进程打印其父进程ID时，我们看到了一个潜在的竞争条件。如果第二个子进程在第一个子进程之前运行，则其父进程将会是第一个子进程。但是，如果第一个子进程先运行，并有足够的时间到达并执行exit，则第二个子进程的父进程就是init。即使在程序中调用sleep，这也不会保证什么。如果系统负载很重，那么在第二个子进程从sleep返回时，可能第一个子进程还没有得到机会运行。这种形式的问题很难排除，因为在大部分时间，这种问题并不会出现。

如果一个进程希望等待一个子进程终止，则它必须调用一种wait函数。如果一个进程要等待其父进程终止（如程序清单8-5中一样），则可使用下列形式的循环：

```
while (getppid() != 1)
    sleep(1);
```

这种形式的循环（称为轮询（polling））的问题是它浪费了CPU时间，因为调用者每隔1秒都被唤醒，然后进行条件测试。

为了避免竞争条件和轮询，在多个进程之间需要有某种形式的信号发送和接收的方法。在UNIX中可以使用信号机制，在10.16节将说明它在解决此方面问题的一种用法。也可使用各种形式的进程间通信（IPC），在第15和17章将对此进行讨论。

在父、子进程的关系中，常常出现下述情况。在调用fork之后，父、子进程都有一些事情要做。例如，父进程可能要用子进程ID更新日志文件中的一个记录，而子进程则可能要为父进程创建一个文件。在本例中，要求每个进程在执行完它的一套初始化操作后要通知对方，并且在继续运行之前，要等待另一方完成其初始化操作。这种方案可以用代码描述如下：

```
#include   "apue.h"
```

```
TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {            /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */
WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

228

假定在头文件apue.h中定义了各个需要使用的变量。5个例程TELL_WAIT、TELL_ PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD可以是宏，也可以是函数。

在后面几章中会说明实现这些TELL和WAIT例程的不同方法：10.16节中说明使用信号的一种实现，程序清单15-3说明使用管道的一种实现。下面先看一个使用这5个例程的实例。

程序清单8-6输出两个字符串：一个由子进程输出，另一个由父进程输出。因为输出依赖于内核使这两个进程运行的顺序及每个进程运行的时间长度，所以该程序包含了一个竞争条件。

程序清单8-6　具有竞争条件的程序

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;
```

```
    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

在程序中将标准输出设置为不带缓冲的，于是每个字符输出都需调用一次write。本例的目的是使内核能尽可能地在两个进程之间进行多次切换，以便演示竞争条件。（如果不这样做，可能也就决不会见到下面所示的输出。没有看到具有错误的输出并不意味着竞争条件不存在，这只是意味着在此特定的系统上未能见到它。）下面的实际输出说明该程序的运行结果是可以改变的。

229

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

修改程序清单8-6，以使用TELL和WAIT函数，于是形成了程序清单8-7。行首标以+号的行是新增加的行。

程序清单8-7 修改程序清单8-6以避免竞争条件

```
  #include "apue.h"

  static void charatatime(char *);

  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }

  static void
  charatatime(char *str)
  {
      char    *ptr;
      int     c;

      setbuf(stdout, NULL);           /* set unbuffered */
      for (ptr = str; (c = *ptr++) != 0; )
          putc(c, stdout);
  }
```

230 运行此程序则能得到所预期的输出；两个进程的输出不再交叉混合。

程序清单8-7是使父进程先运行。如果将fork之后的行改变成

```
} else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();          /* child goes first */
    charatatime("output from parent\n");
}
```

则子进程先运行。习题8.3继续这一实例。□

8.10 exec函数

8.3节曾提及用fork函数创建子进程后，子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程执行的程序完全替换为新程序，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用一个全新的程序替换了当前进程的正文、数据、堆和栈段。

有6种不同的exec函数可供使用，它们常常被统称为exec函数。这些exec函数使得UNIX进程控制原语更加完善。用fork可以创建新进程，用exec可以执行新程序。exit函数和两个wait函数处理终止和等待终止。这些是我们需要的基本的进程控制原语。在后面各节中将使用这些原语构造另外一些如popen和system之类的函数。

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);
```

231 6个函数返回值：若出错则返回-1，若成功则不返回值

这些函数之间的第一个区别是前4个取路径名作为参数，后两个则取文件名作为参数。当指定*filename*作为参数时：

- 如果*filename*中包含/，则将其视为路径名。
- 否则就按PATH环境变量，在它所指定的各目录中搜寻可执行文件。

PATH变量包含了一张目录表（称为路径前缀），目录之间用冒号（:）分隔。例如，*name* = *value*环境字符串

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

指定在4个目录中进行搜索。最后的路径前缀表示当前目录。（零长前缀也表示当前目录。在*value*的开始处可用:表示，在行中间则要用::表示，在行尾则以:表示。）

> 出于安全性方面的考虑，有些人要求在搜索路径中决不要包括当前目录。请参见Garfinkel等[2003]。

如果execlp或execvp使用路径前缀中的一个找到了一个可执行文件，但是该文件不是由连接编辑器产生的机器可执行文件，则认为该文件是一个shell脚本，于是试着调用/bin/sh，并以该*filename*作为shell的输入。

第二个区别与参数表的传递有关（l表示list，v表示矢量vector）。函数execl、execlp和execle要求将新程序的每个命令行参数都说明为一个单独的参数。这种参数表以空指针结尾。对于另外三个函数（execv、execvp和execve），则应先构造一个指向各参数的指针数组，然后将该数组地址作为这三个函数的参数。

在使用ISO C原型之前，对execl、execle和execlp三个函数表示命令行参数的一般方法是

```
char *arg0, char *arg1, ..., char *argn, (char *)0
```

应当特别指出的是：在最后一个命令行参数之后跟了一个空指针。如果用常数0来表示一个空指针，则必须将它强制转换为一个字符指针，否则将它解释为整型参数。如果一个整型数的长度与char *的长度不同，那么exec函数的实际参数就将出错。

最后一个区别与向新程序传递环境表相关。以e结尾的两个函数（execle和execve）可以传递一个指向环境字符串指针数组的指针。其他四个函数则使用调用进程中的environ变量为新程序复制现有的环境（回忆7.9节及表7-2中对环境字符串的讨论。其中曾提及如果系统支持setenv和putenv这样的函数，则可更改当前环境和后面生成的子进程的环境，但不能影响父进程的环境）。通常，一个进程允许将其环境传播给其子进程，但有时也有这种情况，即进程想要为子进程指定某一个确定的环境。例如，在初始化一个新登录的shell时，login程序通常创建一个只定义少数几个变量的特殊环境，而在我们登录时，可以通过shell启动文件，将其他变量加到环境中。 232

在使用ISO C原型之前，execle的参数是

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

从中可见，最后一个参数是指向环境字符串的各字符指针构成的数组的地址。而在ISO C原型中，所有命令行参数、空指针和*envp*指针都用省略号（...）表示。

这6个exec函数的参数很难记忆。函数名中的字符会给我们一些帮助。字母p表示该函数取*filename*作为参数，并且用PATH环境变量寻找可执行文件。字母l表示该函数取一个参数表，它与字母v互斥。v表示该函数取一个*argv*[]矢量。最后，字母e表示该函数取*envp*[]数组，而不使用当前环境。表8-6显示了这6个函数之间的区别。

表8-6 6个exec函数之间的区别

函数	*pathname*	*filename*	参数表	*argv*[]	environ	*envp*[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
名字中的字母		p	l	v		e

每个系统对参数表和环境表的总长度都有一个限制。在2.5.2节和表2-8中，这种限制是由ARG_MAX给出的。在POSIX.1系统中，此值至少是4096字节。当使用shell的文件名扩充功能产生一个文件名表时，可能会受到此值的限制。例如，命令

```
grep getrlimit /usr/share/man/*/*
```

在某些系统上可能产生下列形式的shell错误：

```
Argument list too long
```

由于历史原因，在早期的系统V实现中，此限制值是5120字节。在早期BSD系统中，此限制值是20 480字节。在当前系统中，此限制值要大得多。（见程序清单2-2的输出，限制值列出在表2-12中。）

233

为了摆脱对参数表长度的限制，我们可以使用xargs(1)命令，将长参数表分解成几部分。为了寻找在我们所用系统手册中的getrlimit，我们可以用

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

如果所用的系统手册是压缩过的，则可使用

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

对于find命令，我们使用选项-type f限制输出列表只包含普通文件。这样做的原因是，grep命令不能在目录中搜索模式，我们也想避免不必要的出错消息。

前面曾提及在执行exec后，进程ID没有改变。除此之外，执行新程序的进程还保持了原进程的下列特征：

- 进程ID和父进程ID。
- 实际用户ID和实际组ID。
- 附加组ID。
- 进程组ID。
- 会话ID。
- 控制终端。
- 闹钟尚余留的时间。
- 当前工作目录。
- 根目录。
- 文件模式创建屏蔽字。
- 文件锁。
- 进程信号屏蔽。
- 未处理信号。
- 资源限制。
- tms_utime、tms_stime、tms_cutime以及tms_cstime值。

对打开文件的处理与每个描述符的执行时关闭（close-on-exec）标志值有关。见图3-1以及3.14节中对FD_CLOEXEC的说明，进程中每个打开描述符都有一个执行时关闭标志。若此标志设置，则在执行exec时关闭该描述符，否则该描述符仍打开。除非特地用fcntl设置了该标志，否则系统的默认操作是在执行exec后仍保持这种描述符打开。

POSIX.1明确要求在执行exec时关闭打开的目录流（见4.21节中所述的opendir函数）。这通常是由opendir函数实现的，它调用fcntl函数为对应于打开目录流的描述符设置执行时

关闭标志。 234

注意，在执行exec前后实际用户ID和实际组ID保持不变，而有效ID是否改变则取决于所执行程序文件的设置用户ID位和设置组ID位是否设置。如果新程序的设置用户ID位已设置，则有效用户ID变成程序文件所有者的ID，否则有效用户ID不变。对组ID的处理方式与此相同。

在很多UNIX实现中，这6个函数中只有execve是内核的系统调用。另外5个只是库函数，它们最终都要调用该系统调用。这6个函数之间的关系示于图8-2中。

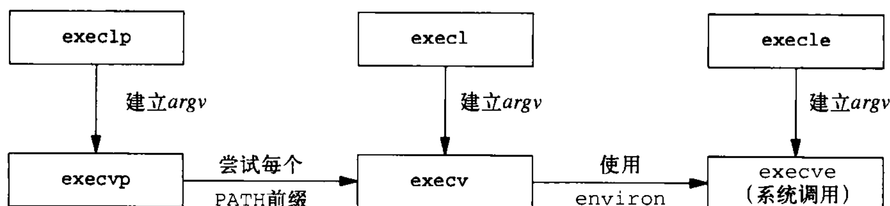


图8-2 6个exec函数之间的关系

在这种安排中，库函数execlp和execvp使用PATH环境变量，查找第一个包含名为filename的可执行文件的路径名前缀。

实例

程序清单8-8演示了exec函数。

程序清单8-8 exec函数实例

```
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

235

在该程序中先调用execle，它要求一个路径名和一个特定的环境。下一个调用的是execlp，它用一个文件名，并将调用者的环境传送给新程序。execlp在这里能够工作的原因是因为目录/home/sar/bin是当前路径前缀之一。注意，我们将第一个参数（新程序中的argv[0]）设置为路径名的文件名分量。某些shell将此参数设置为完整的路径名。这只是一个惯例。我们可将argv[0]设置为任何字符串。当login命令执行shell时就是这样做的。在执行shell之前，login在argv[0]之前加一个/作为前缀，这向shell指明它是作为登录shell被调用的。登录shell将执行启动配置文件（start-up profile）命令，而非登录shell则不会执行这些命令。

程序清单8-8中要执行两次的程序echoall示于程序清单8-9中。这是一个很普通的程序，它回送其所有命令行参数及全部环境表。

程序清单8-9　回送所有命令行参数和所有环境字符串

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)    /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

236

执行程序清单8-8得到

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                                其中有47行没有显示
HOME=/home/sar
```

注意，shell提示符出现在第二个exec打印argv[0]之前。这是因为父进程并不等待该子进程结束。 □

8.11　更改用户ID和组ID

在UNIX系统中，特权（例如能改变当前日期的表示法以及访问控制（例如，能否读、写一特定文件））是基于用户和组ID的。当程序需要增加特权，或需要访问当前并不允许访问的资源时，我们需要更换自己的用户ID或组ID，使得新ID具有合适的特权或访问权限。与此类似，

当程序需要降低其特权或阻止对某些资源的访问时，也需要更换用户ID或组ID，从而使新ID不具有相应特权或访问这些资源的能力。

一般而言，在设计应用程序时，我们总是试图使用*最小特权*（least privilege）模型。依照此模型，我们的程序应当只具有为完成给定任务所需的最小特权。这减少了安全性受到损害的可能性，这种安全性损害是由恶意用户试图哄骗我们的程序以未预料的方式使用特权所造成的。

可以用setuid函数设置实际用户ID和有效用户ID。与此类似，可以用setgid函数设置实际组ID和有效组ID。

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

两个函数返回值：若成功则返回0，若出错则返回−1

关于谁能更改ID有若干规则。现在先考虑有关改变用户ID的规则（我们关于用户ID所说明的一切都适用于组ID）。

237

(1) 若进程具有超级用户特权，则setuid函数将实际用户ID、有效用户ID，以及保存的设置用户ID设置为*uid*。

(2) 若进程没有超级用户特权，但是*uid*等于实际用户ID或保存的设置用户ID，则setuid只将有效用户ID设置为*uid*。不改变实际用户ID和保存的设置用户ID。

(3) 如果上面两个条件都不满足，则将errno设置为EPERM，并返回−1。

在这里假定_POSIX_SAVED_IDS为真。如果没有提供这种功能，则上面所说的关于保存的设置用户ID部分都无效。

> 在POSIX.1 2001版中，保存的ID是强制性特征。而在较早的版本中，它们是可选的。为了弄清楚某种实现是否支持这一特征，应用程序在编译时可以测试常量_POSIOX_SAVED_IDS，或者在运行时以_SC_SAVED_IDS参数调用sysconf函数。

关于内核所维护的三个用户ID，还要注意下列几点：

(1) 只有超级用户进程可以更改实际用户ID。通常，实际用户ID是在用户登录时，由login(1)程序设置的，而且永远不会改变它。因为login是一个超级用户进程，当它调用setuid时，会设置所有三个用户ID。

(2) 仅当对程序文件设置了设置用户ID位时，exec函数才会设置有效用户ID。如果设置用户ID位没有设置，则exec函数不会改变有效用户ID，而将其维持为原先值。任何时候都可以调用setuid，将有效用户ID设置为实际用户ID或保存的设置用户ID。自然，不能将有效用户ID设置为任意随机值。

(3) 保存的设置用户ID是由exec复制有效用户ID而得来的。如果设置了文件的设置用户ID位，则在exec根据文件的用户ID设置了进程的有效用户ID以后，就将这个副本保存起来。

表8-7列出了改变这三个用户ID的不同方法。

注意，8.2节中所述的getuid和geteuid函数只能获得实际用户ID和有效用户ID的当前值。我们不能获得所保存的设置用户ID的当前值。

表8-7　改变三个用户ID的不同方法

ID	exec		setuid (*uid*)	
	设置用户ID位关闭	设置用户ID位打开	超级用户	非特权用户
实际用户ID	不变	不变	设为*uid*	不变
有效用户ID	不变	设置为程序文件的用户ID	设为*uid*	设为*uid*
保存的设置用户ID	从有效用户ID复制	从有效用户ID复制	设为*uid*	不变

238

实例

为了说明保存的设置用户ID特征的用法，先观察一个使用该特征的程序。我们所观察的是man(1)程序，它用于显示联机手册页。man程序可被安装为设置用户ID或设置组ID程序，man程序文件的所有者及他所属的组通常是为man自身保留的用户或组。man程序可以被构造为读以及可能重写文件，通过配置文件（通常是/etc/man.config或/etc/manpath.config）或者使用命令行的选项选择读、写文件的位置。

man程序可能需要执行许多其他命令，以处理包含需显示手册页的文件。为了防止被欺骗运行错误的命令或重写错误的文件，man命令不得不在两种权限之间切换：运行man命令用户的权限，以及拥有man可执行文件用户的权限。下面列出了其工作步骤：

(1) man程序文件是由名为man的用户拥有的，并且其设置用户ID位已设置。当我们exec此程序时，则关于用户ID得到

实际用户ID＝我们的用户ID
有效用户ID＝man
保存的设置用户ID＝man

(2) man程序访问需要的配置文件和手册页。这些文件是由名为man的用户所拥有的，因为有效用户ID是man，所以可以访问这些文件。

(3) 在man代表我们运行任一命令之前，它调用setuid(getuid())。因为我们不是超级用户进程，所以这仅仅会改变有效用户ID。此时得到

实际用户ID＝我们的用户ID（未改变）
有效用户ID＝我们的用户ID
保存的设置用户ID＝man（未改变）

现在，man进程是以我们的用户ID作为其有效用户ID而运行。这就意味着能访问的只有我们通常可以访问的，而没有额外的权限。它可以代表我们安全地执行任一过滤器程序（filter）。

(4) 当执行完过滤器操作后，man调用setuid（*euid*），其中*euid*是用户man的数值用户ID（man调用geteuid，得到用户man的用户ID，然后将其保存起来）。因为setuid的参数等于保存的设置用户ID，所以这种调用是许可的（这就是为什么需要保存的设置用户ID的原因）。现在得到

实际用户ID＝我们的用户ID（未改变）
有效用户ID＝man
保存的设置用户ID＝man（未改变）

(5) 因为man程序的有效用户ID是man，所以现在它可对其文件进行操作。

239

以这种方式使用保存的设置用户ID，于是在进程的开始和结束部分就可以使用由于程序文

件的设置用户ID而得到的额外特权。但是，进程在其运行的大部分时间内只具有普通的权限。如果进程不能在其结束时切换回保存的设置用户ID，那么就不得不在全部运行时间都保持额外的权限（这可能会造成麻烦）。

下面来看一看如果在man运行时为我们生成一个shell进程（先fork，然后exec），这将发生什么？因为实际用户ID和有效用户ID都是我们的普通用户ID（上面的第3步），所以该shell没有额外权限。它不能存取man运行时设置成man的保存的设置用户ID，因为该shell所保存的设置用户ID是由exec复制有效用户ID而得到的。所以在执行exec的子进程中，所有三个用户ID都是我们的普通用户ID。

如果程序文件是设置用户ID为root，那么我们关于man如何使用setuid所做的说明是不正确的。因为以超级用户特权调用setuid就会设置所有三个用户ID。要使上述实例按我们所说明的进行工作，只需setuid仅设置有效用户ID。 □

1. setreuid和setregid函数

历史上，BSD支持setregid函数，其功能是交换实际用户ID和有效用户ID的值。

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

两个函数返回值：若成功则返回0，若出错则返回-1

如若其中任一参数的值为-1，则表示相应的ID应当保持不变。

相关规则很简单：一个非特权用户总能交换实际用户ID和有效用户ID。这就允许一个设置用户ID程序转换成只具有用户的普通权限，以后又可再次转换回设置用户ID所得到的额外权限。POSIX.1引入了保存的设置用户ID特征后，其规则也相应加强，它允许一个非特权用户将其有效用户ID设置为保存的设置用户ID。

seteuid和setregid两个函数都是Single UNIX Specification的XSI扩展。因此，预期所有UNIX系统实现都将提供对它们的支持。

4.3BSD并没有上面所说的保存的设置用户ID特征。它用setreuid和setregid来代替。这就允许一个非特权用户交换这两个用户ID的值，但是要知道，当使用此特征的程序生成shell进程时，它必须在exec之前，先将实际用户ID设置为普通用户ID。如果不这样做的话，那么实际用户ID就可能是具有特权的（由setreuid的交换操作造成），然后shell进程可能会调用setreuid交换两个用户ID值并取得更多权限。作为一个保护性的解决这一问题的编程措施，程序在子进程调用exec之前，将子进程的实际用户ID和有效用户ID都设置成普通用户ID。

240

2. seteuid和setegid函数

POIX.1包含了两个函数seteuid和setegid。它们类似于setuid和setgid，但只更改有效用户ID和有效组ID。

```
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

两个函数返回值：若成功则返回0，若出错则返回-1

一个非特权用户可将其有效用户ID设置为其实际用户ID或其保存的设置用户ID。对于一个特权用户，则可将有效用户ID设置为*uid*。(这有别于setuid函数，它会更改所有三个用户ID。)

图8-3给出了本节所述的修改三个不同用户ID的各个函数。

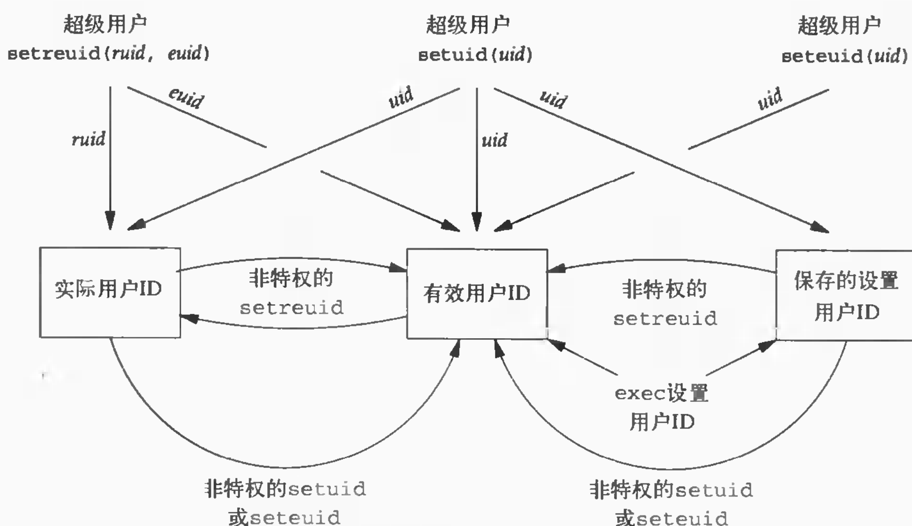


图8-3　设置不同用户ID的各函数

3. 组ID

本章中所说明的一切都以类似方式适用于各个组ID。附加组ID不受setgid、setregid或setegid函数的影响。

241

8.12　解释器文件

所有现今的UNIX系统都支持解释器文件（interpreter file）。这种文件是文本文件，其起始行的形式是：

```
#! pathname [ optional-argument ]
```

感叹号和pathname之间的空格是可选的。最常见的解释器文件以下列行开始：

```
#!/bin/sh
```

*pathname*通常是绝对路径名，对它不进行什么特殊的处理（即不使用PATH进行路径搜索）。对这种文件的识别是由内核作为exec系统调用处理的一部分来完成的。内核使调用exec函数的进程实际执行的并不是该解释器文件，而是该解释器文件第一行中*pathname*所指定的文件。一定要将解释器文件（文本文件，它以#!开头）和解释器（由该解释器文件第一行中的*pathname*指定）区分开来。

要知道很多系统对解释器文件的第一行有长度限制。这些限制包括#!、*pathname*、可选参数、终止换行符以及空格数。

> 在FreeBSD 5.2.1中，该限制是128字节。Mac OS X 10.3将此扩展为512字节。Linux 2.4.22支持该限制为127字节，而Solaris 9设置的限制是1023字节。

让我们观察一个实例，从中可了解当被执行的文件是解释器文件时，内核如何处理exec函数的参数及该解释器文件第一行的可选参数。程序清单8-10调用exec执行一个解释器文件。

程序清单8-10 执行一个解释器文件的程序

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0)  /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

242

下面先显示要被执行的该解释器文件的内容（只有一行），接着是运行程序清单8-10的结果。

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

程序echoarg（解释器）回送每一个命令行参数（它就是程序清单7-3）。注意，当内核exec该解释器（/home/sar/bin/echoarg）时，argv[0]是该解释器的*pathname*，argv[1]是解释器文件中的可选参数，其余参数是*pathname*（/home/sar/bin/testinterp），以及程序清单8-10中调用execl的第二个和第三个参数（myarg1和MY ARG2）。调用execl时的argv[1]和argv[2]已右移了两个位置。注意，内核取execl调用中的*pathname*而非第一个参数（testinterp），因为一般而言，*pathname*包含了比第一个参数更多的信息。□

在解释器*pathname*后可跟随可选参数。如果一个解释器程序支持-f选项，那么在*pathname*后经常使用的就是-f。例如，可以以下列方式执行awk(1)程序：

```
awk -f myfile
```

它告诉awk从文件myfile中读awk程序。

在系统V派生的很多系统中，常包含有awk语言的两个版本。awk常常被称为“老awk”，它是与V7一起分发的原始版本。nawk（新awk）包含了很多增强功能，对应于在Aho、Kernighan和Weinberger[1988]中说明的语言。此新版本提供了对命令行参数的存取，这是下面的示例所需的。Solaris 9提供了两个版本。

POSIX 1003.2标准现在是Single UNIX Specification中基本POSIX.1规范的一部分。在该标准中，awk程序是其中的一个实用程序。该实用程序的基础也是Aho、Kernighan和Weinbeger[1988]中所描述的语言。

Mac OS X 10.3中的awk版本基于贝尔实验室版本，Lucent已将其放在公共域（public domain）中。FreeBSD 5.2.1和Linux 2.4.22提供GNU awk（称为gawk），它链接至名字awk。gawk版本遵循POSIX标准，但也包括了一些扩展。因为贝尔实验室的awk版本和gawk比较新，所以较之nawk或老版本的awk更受欢迎。（贝尔实验室的awk版本可从http://cm.bell-labs.com/cm/cs/awkbook/index.html取用。）

243

在解释器文件中使用-f选项，可以写成

```
#!/bin/awk -f
(在此解释器文件中后随awk程序)
```

例如，程序清单8-11示出了在/usr/local/bin/awkexample中的一个解释器文件。

程序清单8-11 作为解释器文件的awk程序

```
#!/bin/awk -f
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

如果路径前缀之一是/usr/local/bin，则可以用下列方式执行程序清单8-11（假定我们已打开了该文件的执行位）：

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

执行/bin/awk时，其命令行参数是

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

解释器文件的路径名（/usr/local/bin/awkexample）被传送给解释器。因为不能期望该解释器（在本例中是/bin/awk）会使用PATH变量定位该解释器文件，所以只传送其路径名中的文件名是不够的，所以要将解释器文件完整的路径名传送给解释器。当awk读解释器文件时，因为#是awk的注释字符，所以它会忽略第一行。

可以用下列命令验证上述命令行参数：

```
$ /bin/su                              成为超级用户
Password:                              输入超级用户口令
# mv /bin/awk /bin/awk.save            保存原先的程序
# cp /home/sar/bin/echoarg /bin/awk    暂时替换它
# suspend                              用作业控制挂起超级用户shell
[1] + Stopped          /bin/su
$ awkexample file1 FILENAME2 f3
```

```
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                     用作业控制恢复超级用户shell
/bin/su
# mv /bin/awk.save /bin/awk              恢复原先的程序
# exit                                   终止超级用户shell
```

244

在此示例中，解释器的-f选项是必需的。正如前述，它告诉awk在什么地方找到awk程序。如果从解释器文件中删除-f选项，则在试图运行该解释器文件时，通常输出一条出错消息。该出错消息的精确文本可能有所不同，这取决于解释器文件存放在何处，以及其余参数是否表示现有文件等。因为在这种情况下命令行参数是：

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

于是awk企图将字符串/usr/local/bin/awkexample解释为一个awk程序。如果不能向解释器传递至少一个可选参数（在本例中是-f），那么这些解释器文件只有对shell才是有用的。 □

是否一定需要解释器文件呢？那也不完全如此。但是它们确实使用户得到效率方面的好处，其代价是内核的额外开销（因为识别解释器文件的是内核）。由于下述理由，解释器文件是有用的：

(1) 有些程序是用某种语言编写的脚本，解释器文件可将这一事实隐藏起来。例如，为了执行程序清单8-11，只需使用下列命令行：

awkexample *optional-arguments*

而并不需要知道该程序实际上是一个awk脚本，否则就要以下列方式执行该程序：

awk -f awkexample *optional-arguments*

(2) 解释器脚本在效率方面也提供了好处。再考虑一下前面的例子。仍旧隐藏该程序是一个awk脚本的事实，但是将其包装在一个shell脚本中：

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

这种解决方案的问题是要求做更多的工作。首先，shell读此命令，然后试图execlp此文件名。因为shell脚本是一个可执行文件，但却不是机器可执行的，于是返回一个错误，execlp就认为该文件是一个shell脚本（它实际上就是这种文件）。然后执行/bin/sh，并以该shell脚本的路径名作为其参数。shell正确地执行我们的shell脚本，但是为了运行awk程序，它会调用fork、exec和wait。于是，用一个shell脚本代替解释器脚本需要更多的开销。

(3) 解释器脚本使我们可以使用除/bin/sh以外的其他shell来编写shell脚本。当execlp找到一个非机器可执行的可执行文件时，它总是调用/bin/sh来解释执行该文件。但是，使用解释器脚本，则可编写成：

245

```
#!/bin/csh
(在解释器文件中后接C shell脚本)
```

再一次，我们也可将此放在一个/bin/sh脚本中（然后由其调用C shell），但是要有更多的开销。

如果三个shell和awk没有用#作为注释符，则上述方式无效。

8.13 system函数

在程序中执行一个命令字符串很方便。例如，假定要将时间和日期放到某一个文件中，则可使用6.10节中说明的函数实现这一点。调用time得到当前日历时间，接着调用localtime将日历时间转换为年、月、日、时、分、秒、周日形式，然后调用strftime对上面的结果进行格式化处理，最后将结果写到文件中。但是用下面的system函数则更容易做到这一点。

```
system("date > file");
```

ISO C定义了system函数，但是其操作对系统的依赖性很强。POSIX.1包括了system接口，它扩展了ISO C定义，以描述system在POSIX.1环境中的运行行为。

```
#include <stdlib.h>

int system(const char *cmdstring);
```

返回值：（见下）

如果*cmdstring*是一个空指针，则仅当命令处理程序可用时，system返回非0值，这一特征可以确定在一个给定的操作系统上是否支持system函数。在UNIX中，system总是可用的。

因为system在其实现中调用了fork、exec和waitpid，因此有三种返回值：

(1) 如果fork失败或者waitpid返回除EINTR之外的出错，则system返回−1，而且errno中设置了错误类型值。

(2) 如果exec失败（表示不能执行shell），则其返回值如同shell执行了exit(127)一样。

(3) 否则所有三个函数（fork、exec和waitpid）都执行成功，并且system的返回值是shell的终止状态，其格式已在waitpid中说明。

> 如果waitpid由一个捕捉到的信号中断，则某些早期的system实现都返回错误类型值EINTR，但是，因为没有可用的清理策略能让应用程序从这种错误类型中恢复，所以POSIX后来增加了下列要求：在这种情况下system不返回一个错误。（10.5节中将讨论被中断的系统调用。）

246

程序清单8-12是system函数的一种实现。它对信号没有进行处理。10.18节中将修改此函数使其进行信号处理。

程序清单8-12　system函数（没有信号处理）

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)   /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
```

```
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

shell的-c选项告诉shell程序取下一个命令行参数（在这里是*cmdstring*）作为命令输入（而不是从标准输入或从一个给定的文件中读命令）。shell对以null字符终止的命令字符串进行语法分析，将它们分成命令行参数。传递给shell的实际命令字符串可以包含任一有效的shell命令。例如，可以用<和>重定向输入和输出。

如果不使用shell执行此命令，而是试图由我们自己去执行它，那么将相当困难。首先，我们必须用execlp而不是execl，像shell那样使用PATH变量。我们必须将null结尾的命令字符串分成各个命令行参数，以便调用execlp。最后，我们也不能使用任何一个shell元字符。

注意，我们调用_exit而不是exit。这是为了防止任一标准I/O缓冲区（这些缓冲区会在fork中由父进程复制到子进程）在子进程中被冲洗。

247

用程序清单8-13对system的这种版本进行了测试（pr_exit函数定义在程序清单8-3中）。

程序清单8-13 调用system函数

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    int     status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

运行程序清单8-13得到

```
$ ./a.out
Sun Mar 21 18:41:32 EST 2004
normal termination, exit status = 0     对于date
```

```
sh: nosuchcommand: command not found
normal termination, exit status = 127    对于无此种命令
sar      :0      Mar 18 19:45
sar      pts/0   Mar 18 19:45 (:0)
sar      pts/1   Mar 18 19:45 (:0)
sar      pts/2   Mar 18 19:45 (:0)
sar      pts/3   Mar 18 19:45 (:0)
normal termination, exit status = 44     对于exit
```

使用system而不是直接使用fork和exec的优点是：system进行了所需的各种出错处理，以及各种信号处理（在10.18节中的system函数的下一个版本中）。

在UNIX的早期版本中，包括SVR3.2和4.3BSD，都没有waitpid函数，于是父进程用下列形式的语句等待子进程：

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

如果调用system的进程在调用它之前已经生成它自己的子进程，那么将引起问题。因为上面的while语句一直循环执行，直到由system产生的子进程终止才停止，如果不是用pid标识的任一子进程在pid子进程之前终止，则它们的进程ID和终止状态都会被while语句丢弃。实际上，由于wait不能等待一个指定的进程以及其他一些原因，POSIX.1 Rationale才定义了waitpid函数。如果不提供waitpid函数，popen和pclose函数也会发生同样的问题（见15.3节）。

248

设置用户ID程序

如果在一个设置用户ID程序中调用system，那么发生什么呢？这是一个安全性方面的漏洞，决不应当这样做。程序清单8-14是一个简单程序，它只是对其命令行参数调用system函数。

程序清单8-14　用system执行命令行参数

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

将此程序编译成可执行文件tsys。

程序清单8-15是另一个简单程序，它打印其实际和有效用户ID。

程序清单8-15　打印实际和有效用户ID

```
#include "apue.h"

int
main(void)
```

```
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

将此程序编译成可执行文件printuids。运行这两个程序，得到下列结果：

249

```
$ tsys printuids                          正常执行，无特权
real uid = 205, effective uid = 205
normal termination, exit status = 0
$ su                                      成为超级用户
Password:                                 输入超级用户口令
# chown root tsys                         更改所有者
# chmod u+s tsys                          增加设置用户ID
# ls -l tsys                              检验文件权限和所有者
-rwsrwxr-x  1 root    16361 Mar 16 16:59 tsys
# exit                                    退出超级用户 shell
$ tsys printuids
real uid = 205, effective uid = 0         哎呀！这是一个安全性漏洞
normal termination, exit status = 0
```

我们给予tsys程序的超级用户权限在system中执行了fork和exec之后仍会保持下来。

> 当/bin/sh是bash版本2时，上面的实例不能工作，其原因是：当有效用户ID与实际用户ID不匹配时，bash将有效用户ID设置为实际用户ID。

如果一个进程正以特殊的权限（设置用户ID或设置组ID）运行，它又想生成另一个进程执行另一个程序，则它应当直接使用fork和exec，而且在fork之后、exec之前要改回到普通权限。设置用户ID或设置组ID程序决不应调用system函数。

> 这种警告的一个理由是：system调用shell对命令字符串进行语法分析，而shell使用IFS变量作为其输入字段分隔符。早期的shell版本在被调用时不将此变量恢复为普通字符集。这就允许一个有恶意的用户在调用system之前设置IFS，造成system执行一个不同的程序。

8.14 进程会计

大多数UNIX系统提供了一个选项以进行进程会计（process accounting）处理。启用该选项后，每当进程结束时内核就写一个会计记录。典型的会计记录包含总量较小的二进制数据，一般包括命令名、所使用的CPU时间总量、用户ID和组ID、启动时间等。本节将较详细地说明这种会计记录，这样也使我们得到了一个再次观察进程的机会，以及使用5.9节中介绍的fread函数的机会。

> 任一标准都没有对进程会计进行过说明。于是，所有实现都有令人厌烦的差别。例如，关于I/O的数量，Solaris 9使用的单位是字节，FreeBSD 5.2.1和Mac OS X 10.3使用的单位是块，但又不考虑不同的块长，这使得该计数值并无实际效用。Linux 2.4.22则根本没有维持I/O统计。
>
> 每种实现也都有自己的一套管理命令去处理这种原始的会计数据。例如，Solaris提供了runacct(1m)和acctcom(1)，FreeBSD则提供sa(8)命令处理并汇总原始会计数据。

一个至今没有说明的函数（acct）用于启用和禁用进程会计。唯一使用这一函数的是accton(8)命令（这是碰巧在几种平台上都类似的少数几条命令中的一条）。超级用户执行一个

250

带路径名参数的accton命令启动会计处理。会计记录写到指定的文件中，在FreeBSD和Mac OS X中，该文件通常是/var/account/acct，在Linux中，该文件是/var/account/ pacct，在Solaris中，则是/var/adm/pacct。执行不带任何参数的accton命令可停止会计处理。

会计记录结构定义在头文件<sys/acct.h>中，其形式如下：

```
typedef  u_short comp_t;  /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char    ac_flag;     /* flag (see Figure 8.26) */
  char    ac_stat;     /* termination status (signal & core flag only) */
                       /* (Solaris only) */
  uid_t   ac_uid;      /* real user ID */
  gid_t   ac_gid;      /* real group ID */
  dev_t   ac_tty;      /* controlling terminal */
  time_t  ac_btime;    /* starting calendar time */
  comp_t  ac_utime;    /* user CPU time (clock ticks) */
  comp_t  ac_stime;    /* system CPU time (clock ticks) */
  comp_t  ac_etime;    /* elapsed time (clock ticks) */
  comp_t  ac_mem;      /* average memory usage */
  comp_t  ac_io;       /* bytes transferred (by read and write) */
                       /* "blocks" on BSD systems */
  comp_t  ac_rw;       /* blocks read or written */
                       /* (not present on BSD systems) */
  char    ac_comm[8];  /* command name: [8] for Solaris, */
                       /* [10] for Mac OS X, [16] for FreeBSD, and */
                       /* [17] for Linux */
};
```

其中，ac_flag成员记录了进程执行期间的某些事件。这些事件见表8-8。

表8-8 会计记录中的ac_flag值

ac_flag	说　明	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
AFORK	进程是由fork产生的，但从未调用exec	•	•	•	•
ASU	进程使用超级用户特权		•	•	•
ACOMPAT	进程使用兼容模式				
ACORE	进程转储core	•	•	•	
AXSIG	进程由信号杀死	•	•	•	
AEXPND	扩展的会计条目				•

会计记录所需的各种数据（如CPU时间、传输的字符数）都由内核保存在进程表中，并在一个新进程被创建时置初值（例如调用fork之后在子进程中）。每次进程终止时都会编写一条会计记录。这就意味着在会计文件中记录的顺序对应于进程终止的顺序，而不是它们启动的顺序。为了确定启动顺序，需要读全部会计文件，并按启动日历时间进行排序。这不是一种很完善的方法，因为日历时间的单位是秒（见1.10节），在给定的那一秒钟可能启动了多个进程。而墙上时钟时间是由时钟滴答（通常，每秒滴答数在60至128之间）给出的。但是我们并不知道进程的终止时间，所知道的只是启动时间和终止顺序。这就意味着，即使墙上时钟时间比启动时间要精确得多，但是仍不能按照会计文件中的数据重构各进程的精确启动顺序。

251

会计记录对应于进程而不是程序。在fork之后，内核为子进程初始化一个记录，而不是在一个新程序被执行时做这项工作。虽然exec并不创建一个新的会计记录，但改变了相应记录中的命令名，并且AFORK标志会被清除。这意味着，如果一个进程顺序执行了三个程序

(A exec B，然后B exec C，最后C exit)，但只会写一条会计记录。该记录中的命令名对应于程序C，但CPU时间是程序A、B、C之和。

实例

为了得到某些会计数据以便查看，我们按图8-4编写了测试程序（见程序清单8-16）。

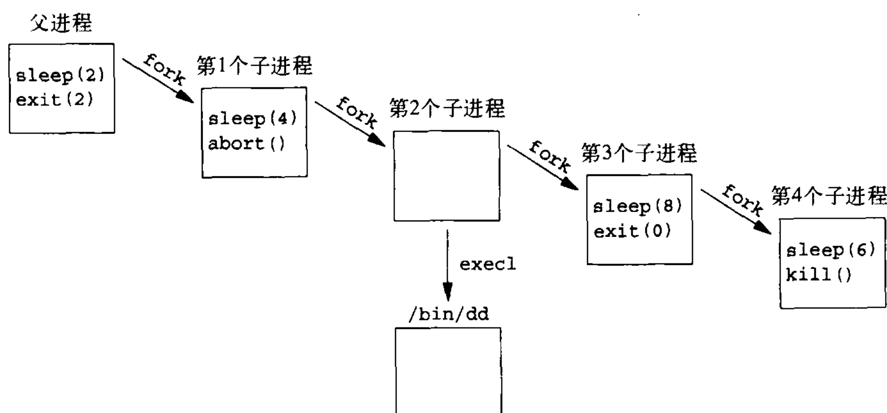


图8-4　会计处理实例的进程结构

该程序调用fork四次。每个子进程做不同的事情，然后终止。

252

程序清单8-16　产生会计数据的程序

```
#include "apue.h"

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

                                /* first child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                /* terminate with core dump */
    }

                                /* second child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
        exit(7);                /* shouldn't get here */
    }

                                /* third child */
```

```
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                /* normal exit */
    }

                                /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL);    /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```

253

在Solaris上运行该测试程序，然后用程序清单8-17从会计记录中选择一些字段并打印出来。

程序清单8-17 打印从系统会计文件中选出的字段

```
#include "apue.h"
#include <sys/acct.h>

#ifdef HAS_SA_STAT
#define FMT "%-*.*s  e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n"
#else
#define FMT "%-*.*s  e = %6ld, chars = %7ld, %c %c %c %c\n"
#endif
#ifndef HAS_ACORE
#define ACORE 0
#endif
#ifndef HAS_AXSIG
#define AXSIG 0
#endif

static unsigned long
compt2ulong(comp_t comptime)    /* convert comp_t to unsigned long */
{
    unsigned long   val;
    int             exp;

    val = comptime & 0x1fff;    /* 13-bit fraction */
    exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
int
main(int argc, char *argv[])
{
    struct acct     acdata;
    FILE            *fp;

    if (argc != 2)
        err_quit("usage: pracct filename");
    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf(FMT, (int)sizeof(acdata.ac_comm),
            (int)sizeof(acdata.ac_comm), acdata.ac_comm,
            compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
#ifdef HAS_SA_STAT
            (unsigned char) acdata.ac_stat,
#endif
            acdata.ac_flag & ACORE ? 'D' : ' ',
            acdata.ac_flag & AXSIG ? 'X' : ' ',
            acdata.ac_flag & AFORK ? 'F' : ' ',
```

```
            acdata.ac_flag & ASU   ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}
```

254

BSD派生的平台不支持ac_flag成员，所以我们在支持该成员的平台上定义了HAS_SA_STAT常量。基于特征而非平台定义的符号常量读起来方便，也使我们易于修改程序，使用的修改方法是，对编译命令增加附加的定义。替代方法可以是使用

```
#if defined(BSD) || defined(MACOS)
```

但是，当将应用程序移植到其他平台上时，这种方法会带来很大的不便。

我们定义了类似的常量以判断该平台是否支持ACORE和AXSIG会计标志。我们不能直接使用这两个标志符号本身，其原因是：在Linux中，它们被定义为enum值，而在#ifdef表达式中不能使用此种类型的值。

为了进行测试，执行下列操作步骤：

(1) 成为超级用户，用accton命令启动会计事务处理。注意，当此命令结束时，会计事务处理已经启动，因此在会计文件中的第一条记录应来自这一命令。

(2) 退出超级用户shell，运行程序清单8-16。这会将6个记录追加到会计文件中（超级用户shell一个，父进程一个，四个子进程各一个）。

在第二个子进程中，execl并不创建一个新进程，所以对第二个进程只有一个会计记录。

(3) 成为超级用户，停止会计事务处理。因为在accton命令终止时已经停止处理会计事务，所以不会在会计文件中增加一个记录。

(4) 运行程序清单8-17，从会计文件中选出字段并打印。

第4步的输出如下所示。在每一行中都对进程加了说明，以便后面讨论。

```
accton    e =    6, chars =      0, stat =   0:     S
sh        e = 2106, chars =  15632, stat =   0:     S
dd        e =    8, chars = 273344, stat =   0:        第二个子进程
a.out     e =  202, chars =    921, stat =   0:        父进程
a.out     e =  407, chars =      0, stat = 134:   F    第一个子进程
a.out     e =  600, chars =      0, stat =   9:   F    第四个子进程
a.out     e =  801, chars =      0, stat =   0:   F    第三个子进程
```

墙上时钟时间值是以每秒滴答数为单位测量的。从表2-12可见，本系统的每秒滴答数是100。例如，在父进程中的sleep(2)对应于202个时钟滴答的墙上时钟时间。对于第一个子进程，sleep(4)变成407个时钟滴答。注意，一个进程休眠的时间总量并不精确。（第10章将返回到sleep函数。）此外，调用fork和exit也需要一些时间。

注意，ac_stat成员并不是进程的真正终止状态。它只是8.6节中讨论的终止状态的一部分。如果进程异常终止，则此字节包含的信息只是核心标志位（一般是最高位）以及信号编号（一般是低7位）。如果进程正常终止，则从会计文件不能得到进程的退出（exit）状态。对于第一个进程，此值是128+6。128是core标志位，6碰巧是此系统信号SIGABRT的值（它是调用abort产生的）。第四个子进程的值是9，它对应于SIGKILL的值。从会计文件的数据中不能了解到，父进程在退出时所用的参数值是2，三个子进程退出时所用的参数值是0。

255

dd进程将文件/etc/termcap复制到第二个子进程中，该文件的长度是136 663字节。而I/O字符数是此值的两倍，其原因是读了136 663字节，然后又写了136 663字节。即使输出到空

设备，仍然会统计I/O字符数。

ac_flag值与我们所预料的相同。除调用execl的第二个子进程以外，其他子进程都设置了F标志。父进程没有设置F标志，其原因是交互式shell调用fork生成父进程，然后父进程执行a.out文件。第一个子进程调用abort，abort产生信号SIGABRT，由此进行了core转储。该进程的X标志和D标志都没有打开，因为Solaris不支持它们；相关信息可从ac_stat字段导出。第四个子进程也因信号而终止，但是SIGKILL信号并不产生core转储，它只是终止该进程。

最后要说明的是：第一个子进程的I/O字符数为0，但是该进程产生了一个core文件。其原因是写core文件所需的I/O并不由该进程负责。

8.15　用户标识

任一进程都可以得到其实际和有效用户ID及组ID。但是有时希望找到运行该程序的用户登录名。我们可以调用getpwuid（getuid()），但是如果一个用户有多个登录名，这些登录名又对应着同一个用户ID，那么又将如何呢？（一个人在口令文件中可以有多个登录项，它们的用户ID相同，但登录shell则不同。）系统通常记录用户登录时使用的名字（见6.8节），用getlogin函数可以获取此登录名。

```
#include <unistd.h>

char *getlogin(void);
```

返回值：若成功则返回指向登录名字符串的指针，若出错则返回NULL

如果调用此函数的进程没有连接到用户登录时所用的终端，则本函数会失败。通常称这些进程为守护进程（daemon），第13章将讨论它们。

给出了登录名，就可用getpwnam在口令文件中查找用户的相应记录，从而确定其登录shell等。

256

为了找到登录名，UNIX系统在历史上一直是调用ttyname函数（见18.9节），然后在utmp文件（见6.8节）中查找匹配项。FreeBSD和Mac OS X将登录名存放在与进程表项相关联的会话结构中，并提供系统调用来获取和存储该登录名。

系统V提供cuserid函数返回登录名。此函数先调用getlogin函数，如果失败则再调用getpwuid（getuid()）。IEEE Std.1003.1-1988说明了cuserid，但是它以有效用户ID而不是实际用户ID来调用。POSIX.1的1990版本删除了cuserid函数。

环境变量LOGNAME通常由login(1)以用户的登录名对其赋初值，并由登录shell继承。但是，用户可以改变环境变量，所以不能使用LOGNAME来确认用户，而应当使用getlogin函数。

8.16　进程时间

在1.10节中说明了我们可以测量的三种时间：墙上时钟时间、用户CPU时间和系统CPU时间。任一进程都可调用times函数以获得它自己及已终止子进程的上述值。

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

返回值：若成功则返回流逝的墙上时钟时间（单位：时钟滴答数），若出错则返回-1

此函数填写由*buf*指向的tms结构，该结构定义如下：

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

注意，此结构没有包含墙上时钟时间的任何测量值。作为替代，times函数返回墙上时钟时间作为其函数值。此值是相对于过去的某一时刻测量的，所以不能用其绝对值，而必须使用其相对值。例如，调用times，保存其返回值。在以后某个时间再次调用times，从新的返回值中减去以前的返回值，此差值就是墙上时钟时间。（一个长期运行的进程可能会使墙上时钟时间溢出，当然这种可能性极小，见习题1.6。）

该结构中两个针对子进程的字段包含了此进程用wait、waitid或waitpid已等待到的各个子进程的值。

所有由此函数返回的clock_t值都用_SC_CLK_TCK（由sysconf函数返回的每秒时钟滴答数，见2.5.4节）变换成秒数。

257

> 大多数实现都提供了getrusage(2)函数，该函数返回CPU时间，以及指示资源使用情况的另外14个值。该函数起源于BSD系统，所以与其他实现相比，BSD派生的实现支持的字段要多一些。

实 例

程序清单8-18将每个命令行参数作为shell命令串执行，对每个命令计时，并打印从tms结构取得的值。

程序清单8-18 时间以及执行所有命令行参数

```
#include "apue.h"
#include <sys/times.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int     i;

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)       /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1)   /* starting values */
        err_sys("times error");
```

```
    if ((status = system(cmd)) < 0)     /* execute command */
        err_sys("system() error");

    if ((end = times(&tmsend)) == -1)       /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");
    printf("  real:  %7.2f\n", real / (double) clktck);
    printf("  user:  %7.2f\n",
      (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf("  sys:   %7.2f\n",
      (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf("  child user:  %7.2f\n",
      (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf("  child sys:   %7.2f\n",
      (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

258

运行此程序，得到：

```
$ ./a.out "sleep 5" "date"

command: sleep 5
  real:     5.02
  user:     0.00
  sys:      0.00
  child user:     0.01
  child sys:      0.00
normal termination, exit status = 0

command: date
Mon Mar 22 00:43:58 EST 2004
  real:     0.01
  user:     0.00
  sys:      0.00
  child user:     0.01
  child sys:      0.00
normal termination, exit status = 0
```

在这两个实例中，子进程中显示的所有CPU时间都是执行shell和命令的子进程所使用的CPU时间。 □

8.17 小结

对于UNIX环境中的高级编程而言，完整地了解UNIX的进程控制是非常重要的。其中必须熟练掌握的只有几个函数——fork、exec族、_exit、wait和waitpid。很多应用程序都使用这些原语。fork原语也给了我们一个了解竞争条件的机会。

本章说明了system函数和进程会计，这也使我们能进一步了解所有这些进程控制函数。

本章还说明了exec函数的另一种变体：解释器文件及其工作方式。理解各种不同的用户ID和组ID（实际、有效和保存的），对编写安全的设置用户ID程序是至关重要的。 259

在了解进程和子进程的基础上，下一章将进一步说明一个进程和其他进程的关系——会话和作业控制。第10章将说明信号机制并以此结束对进程的讨论。

习题

8.1 在程序清单8-2中，如果用exit调用代替_exit调用，那么这可能关闭标准输出，并且printf返回-1。修改该程序验证在你所使用的系统上是否产生此种结果。如果并非如此，你怎样处理才能得到类似结果呢？

8.2 回忆图7-3中典型的存储空间布局。由于对应于每个函数调用的栈帧通常存储在栈中，并且由于调用vfork后，子进程运行在父进程的地址空间中，如果不是在main函数中而是在另一个函数中调用vfork，以后子进程从该函数返回时，将会发生什么情况？编写一段程序对此进行验证，并且画图说明发生了什么。

8.3 当用$ **./a.out**执行程序清单8-7一次时，其输出是正确的。但是若将该程序按下列方式执行多次，则其输出不正确。

```
$ ./a.out ; ./a.out ; ./a.out
output from parent
ooutput from parent
ouotuptut from child
put from parent
output from child
utput from child
```

这将会发生什么？怎样才能更正这种错误？如果使子进程首先输出，还会发生此问题吗？

8.4 在程序清单8-10中，调用execl，指定解释器文件的*pathname*。如果将其改为调用execlp，指定testinterp的*filename*，并且如果目录/home/sar/bin是路径前缀，则运行该程序时，argv[2]的打印输出是什么？

8.5 一个进程怎样才能获得其保存的设置用户ID？

8.6 编写一段程序，创建一个僵死进程，然后调用system执行ps(1)命令以验证该进程是僵死进程。

8.7 8.10节中提及POSIX.1要求在调用exec时关闭打开的目录流。按下列方法对此进行验证：对根目录调用opendir，查看在你的系统上实现的DIR结构，然后打印执行时关闭标志。接着open同一目录读取并打印执行时关闭标志。 260

第9章 进程关系

9.1 引言

在上一章我们已了解到进程之间具有关系。首先，每个进程都有一个父进程（初始的内核进程并无父进程，也可以说其父进程就是它自己）。当子进程终止时，父进程得到通知并能取得子进程的退出状态。在8.6节说明waitpid函数时，我们也提到了进程组，以及如何等待进程组中的任意一个进程终止。

本章将更详细地说明进程组以及POSIX.1引入的会话的概念。还将介绍登录shell（登录时所调用的）和所有从登录shell启动的进程之间的关系。

在说明这些关系时不可能不谈及信号，而讨论信号时又需要很多本章介绍的概念。如果不熟悉UNIX系统信号机制，则可能先要浏览一下第10章。

9.2 终端登录

先说明当我们登录到UNIX系统时所执行的各个程序。在早期的UNIX系统（例如V7）中，用户用哑终端（用硬连接连到主机）进行登录。终端要么是本地的（直接连接）要么是远程的（通过调制解调器连接）。在这两种情况下，登录都经由内核中的终端设备驱动程序。例如，在PDP-11上常用的设备是DH-11和DZ-11。因为连到主机上的终端设备数已经确定，所以同时的登录数也就有了已知的上限。

随着位映射图形终端变成可用，开发出了窗口系统，它向用户提供了与主机系统进行交互的新方式。创建“终端窗口”的应用程序也被开发出来，它仿真了基于字符的终端，使得用户可以用熟悉的方式（即通过shell命令行）与主机交互。

现今，某些平台允许用户在登录后启动一个窗口系统，而另一些平台则自动为用户启动窗口系统。在后一种情况中，用户可能仍然需要登录，这取决于窗口系统是如何配置的（某些窗口系统可配置成自动登录用户）。

我们现在说明的过程用于经由终端登录至UNIX系统。该过程是类似的，而与所使用的终端无关，终端可以是基于字符的终端、仿真简单的基于字符终端的图形终端，或者是运行窗口系统的图形终端。

1. BSD终端登录

在过去30年中，登录过程并没有多少改变。系统管理员创建通常名为/etc/ttys的文件，其中，每个终端设备都有一行，每一行说明设备名和传递给getty程序的参数，例如，参数之一说明了终端的波特率等。当系统自举时，内核创建进程ID为1的进程，也就是init进程。init进程使系统进入多用户状态。init进程读文件/etc/ttys，对每一个允许登录的终端设备，init

调用一次fork，它所生成的子进程则执行（exec）getty程序。图9-1中显示了这些进程。

图9-1中各个进程的实际用户ID和有效用户ID都是0（即它们都具有超级用户特权）。init 262 以空环境执行getty程序。

getty为终端设备调用open函数，以读、写方式将终端打开。如果设备是调制解调器，则open可能会在设备驱动程序中滞留，直到用户拨号调制解调器，并且呼叫被应答。一旦设备被打开，则文件描述符0、1、2就被设置到该设备。然后getty输出"login:"之类的信息，并等待用户键入用户名。如果终端支持多种速度，则getty可以测试特殊字符以便适当地更改终端速度（波特率）。关于getty程序以及有关数据文件（gettytab）的细节，请参阅UNIX系统手册。

当用户键入了用户名后，getty的工作就完成了。然后它以类似于下面的方式调用login程序：

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

（在gettytab文件中可能会有一些选项使其调用其他程序，但系统默认的是login程序）。init以一个空环境调用getty。getty以终端名（例如TERM=foo，其中终端foo的类型取自gettytab文件）和在gettytab中说明的环境字符串为login创建一个环境（envp参数）。-p标志通知login保留传给它的环境，也可将其他环境字符串加到该环境中，但是不要替换它。图9-2显示了login刚被调用后这些进程的状态。

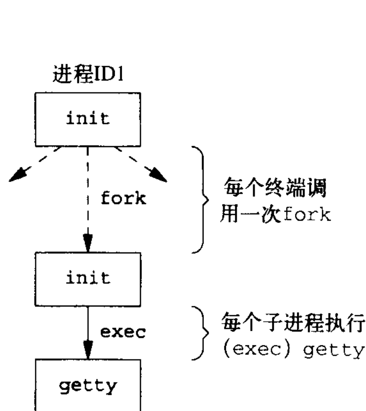


图9-1　init为允许终端登录而调用的进程

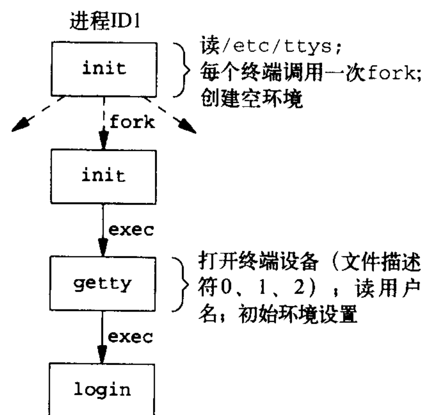


图9-2　调用login后的进程状态

因为最初的init进程具有超级用户特权，所以图9-2中的所有进程都有超级用户特权。图9-2中底部三个进程的进程ID相同，因为进程ID不会因执行exec而改变。并且，除了最初的init进程，所有进程的父进程ID均为1。

login能执行多项工作。因为它得到了用户名，所以能调用getpwnam取得相应用户的口令文件登录项。然后调用getpass(3)以显示提示"Password:"，接着读用户键入的口令 263 （自然，禁止回送用户键入的口令）。它调用crypt(3)将用户键入的口令加密，并与该用户在阴影口令文件中登录项的pw_passwd字段相比较。如果用户几次键入的口令都无效，则login以参数1调用exit表示登录过程失败。父进程（init）了解到子进程的终止情况后，将再次调用fork，其后接着执行getty，对此终端重复上述过程。

这是UNIX系统传统的用户身份验证过程。现代UNIX系统已发展到支持多个身份验证过程。例如，FreeBSD、Linux、Mac OS X以及Solaris都支持被称为PAM（Pluggable Authentication

Module，可插入式身份验证模块）的更加灵活的方案。PAM允许管理员配置使用何种身份验证方法来访问那些使用PAM库编写成的服务。

如果应用程序需验证一用户是否具有适当的权限去执行某个服务，那么我们可以将身份验证机制编写到应用中，或者使用PAM库来得到等价的功能。使用PAM的优点是，管理员可以基于本地策略、针对不同任务配置不同的验证用户身份的方法。

如果用户正确登录，login就将执行如下工作：

- 将当前工作目录更改为该用户的起始目录（chdir）。
- 调用chown改变该终端的所有权，使登录用户成为它的所有者。
- 将对该终端设备的访问权限改变成用户读和写。
- 调用setgid及initgroups设置进程的组ID。
- 用login所得到的所有信息初始化环境：起始目录（HOME）、shell（SHELL）、用户名（USER和LOGNAME），以及一个系统默认路径（PATH）。
- login进程改变为登录用户的用户ID（setuid）并调用该用户的登录shell，如下：

```
execl("/bin/sh", "-sh", (char *)0);
```

argv[0]的第一个字符“-”是一个标志，表示该shell被调用为登录shell。shell可以查看此字符，并相应地修改其启动过程。

login程序实际所做的比上面说的要多。它可选择打印日期消息（message-of-the-day）文件，检查新邮件以及执行其他一些任务。但是考虑到本书的内容，我们主要关心上面所说的功能。

回忆8.11节中对setuid函数的讨论，因为setuid是由超级用户调用的，它更改所有三个用户ID：实际用户ID、有效用户ID和保存的用户ID。login在较早时间调用的setgid对所有三个组ID也有同样效果。

到此为止，登录用户的登录shell开始运行。其父进程ID是init进程ID（进程ID 1），所以当此登录shell终止时，init会得到通知（接到SIGCHLD信号），它会对该终端重复全部上述过程。将登录shell的文件描述符0、1和2设置为终端设备。图9-3显示了这种安排。

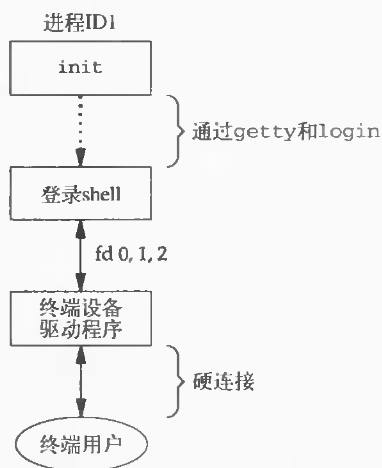


图9-3 为终端登录完成各种设置后的进程安排

264

现在，登录shell读取其启动文件（Bourne shell和Korn shell是：.profile；GNU Bourne-again shell是.bash_profile、.bash_login或.profile；C shell是.cshrc和.login）。这些启动文件通常会改变某些环境变量，加上很多环境变量。例如，很多用户设置他们自己的PATH，常常提示实际终端类型（TERM）。当执行完启动文件后，用户最后得到shell提示符，并能键入命令。

2. Mac OS X终端登录

Mac OS X部分基于FreeBSD，所以其终端登录进程与BSD登录进程的工作步骤相同。但是，Mac OS X一开始提供的就是图形终端。

3. Linux终端登录

Linux的终端登录过程非常类似于BSD。确实，Linux login命令是从4.3BSD login命令

派生而来的。BSD登录过程与Linux登录过程的主要区别在于说明终端配置的方式。

在Linux中，/etc/inittab包含配置信息，它说明了init应当为之启动getty进程的各终端设备，这类似于系统V的方式。依赖于所使用的getty的版本，终端的各种特性要么在命令行上说明（如同使用agetty一样），要么在文件/etc/gettyefs中说明（如同使用mgetty一样）。

265

4. Solaris终端登录

Solaris支持两种形式的终端登录：(a) getty方式，这与上面对BSD所说明的一样；(b) ttymon登录，这是SVR4引入的一种新特性。通常，getty用于控制台，ttymon则用于其他终端登录。

ttymon命令是服务访问设施（Service Access Facility, SAF）的一部分。SAF的目的是用一致的方式对提供系统访问的服务进行管理。（关于SAF的详细信息，参见Rago[1993]的第6章。）按照本书的宗旨，我们只简单说明从init到登录shell之间不同的工作步骤，最后结果则与图9-3中所示相似。init是sac（service access controller，服务访问控制器）的父进程，sac调用fork，然后，当系统进入多用户状态时，其子进程执行ttymon程序。ttymon监控列于配置文件中的所有终端端口，当用户键入登录名时，它调用一次fork。在此之后ttymon的子进程执行login，它向用户发出提示，要求输入口令。一旦完成这一处理，login执行登录用户的登录shell，于是到达了图9-3中所示的位置。一个区别是用户登录shell的父进程现在是ttymon，而在getty登录中，登录shell的父进程则是init。

9.3 网络登录

通过串行终端登录至系统和经由网络登录至系统两者之间的主要（物理上的）区别是：通过网络登录时，终端和计算机之间的连接不是点对点连接。在这种情况下，login只是一种可用的服务，这与其他网络服务（例如FTP或SMTP）的性质相同。

在上一节所述的终端登录中，init知道哪些终端设备可用来进行登录，并为每个设备生成一个getty进程。但是，在网络登录情况下，所有登录都经由内核的网络接口驱动程序（如以太网驱动程序），事先并不知道将会有多少这样的登录。我们不是使一个进程等待每个可能的登录，而是必须等待一个网络连接请求的到达。

为使同一个软件既能处理终端login，又能处理网络login，系统使用了一种称为伪终端（pseudo terminal）的软件驱动程序，它仿真串行终端的运行行为，并将终端操作映射为网络操作，反之亦然。（在第19章，我们将详细说明伪终端。）

1. BSD网络登录

在BSD中，有一个称为inetd的进程（有时称之为因特网超级服务器），它等待大多数网络连接。本节将说明BSD网络登录中所涉及的进程序列。关于这些进程的网络程序设计方面的细节，请参阅Stevens、Fenner和Rudoff [2004]。我们在此不详细说明。

作为系统启动的一部分，init调用一个shell，使其执行shell脚本/etc/rc。由此shell脚本启动一个守护进程inetd。一旦此shell脚本终止，inetd的父进程就变成init。inetd等待TCP/IP连接请求到达主机，而当一个连接请求到达时，它执行一次fork，然后生成的子进程执行适当的程序。

266

我们假定到达了一个针对TELNET服务进程的TCP连接请求。TELNET是使用TCP协议的远程登录应用程序。在另一台主机（它通过某种形式的网络与服务进程的主机相连接）上的用户，或在同一台主机上的用户启动TELNET客户进程，由此启动登录过程：

```
telnet hostname
```

该客户进程打开一个到*hostname*主机的TCP连接，在*hostname*主机上启动的程序被称为TELNET服务进程。然后，客户进程和服务进程之间使用TELNET应用协议通过TCP连接交换数据。所发生的是启动客户进程的用户现在登录到了服务进程所在的主机。（自然，用户需要在服务进程主机上有一个有效的账号）。图9-4显示了在执行TELNET服务进程（称为telnetd）时所涉及的进程序列。

然后，`telnetd`进程打开一个伪终端设备，并用`fork`分成两个进程。父进程处理通过网络连接的通信，子进程则执行`login`程序。父、子进程通过伪终端相连接。在调用`exec`之前，子进程使其文件描述符0、1、2与伪终端相连。如果登录正确，`login`就执行9.2节中所述的同样步骤：更改当前工作目录为起始目录，设置登录用户的组ID和用户ID，以及登录用户的初始环境。然后`login`调用`exec`将其自身替换为登录用户的登录shell。图9-5显示了此时的进程安排。

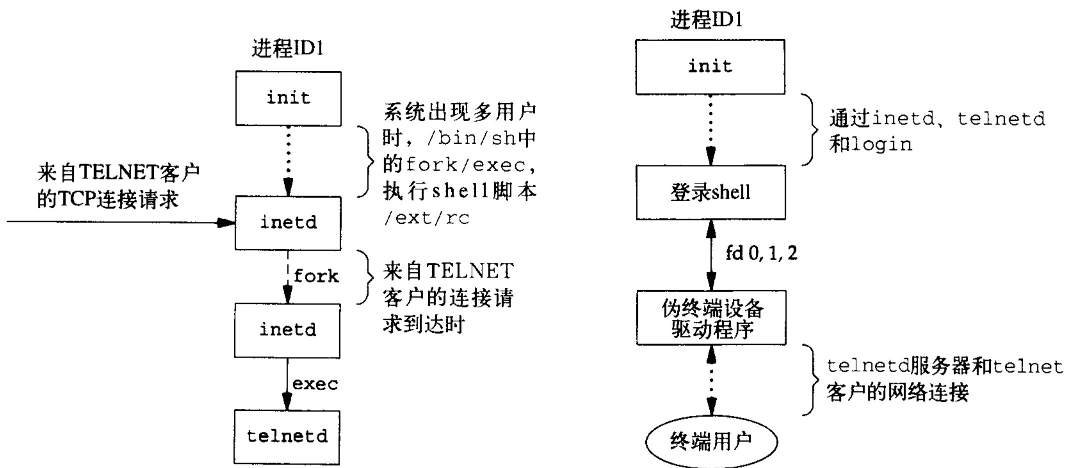


图9-4 执行TELNET服务进程时涉及的进程序列　　图9-5 为网络登录完成各种设置后的进程安排

很明显，在伪终端设备驱动程序和终端实际用户之间有很多事情正在发生。第19章详细说明伪终端时，我们会介绍与这种安排相关的所有进程。

267

需要理解的重点是：当通过终端（见图9-3）或网络（见图9-5）登录时，我们得到一个登录shell，其标准输入、输出和标准出错连接到一个终端设备或者伪终端设备上。在下一节中我们会了解到这一登录shell是一个POSIX.1会话的开始，而此终端或伪终端则是会话的控制终端。

2. Mac OS X网络登录

由于Mac OS X部分基于FreeBSD，因此经由网络登录至Mac OS X系统与BSD系统完全相同。

3. Linux网络登录

除了使用扩展的因特网服务守护进程`xinetd`代替`inetd`进程外，Linux网络登录的其他方面与BSD相同。`xinetd`进程对它所启动的各种服务的控制比`inetd`提供的更加精细。

4. Solaris网络登录

Solaris中的网络登录方案与BSD和Linux中的步骤几乎完全一样。它同样使用了类似于BSD版本的`inetd`服务进程，但是在Solaris中，`inetd`具有一种附加的能力，使其可以在服务访问设施框架下运行，尽管它并没有配置成按此种方式运行。作为替代，`inetd`服务进程由`init`启动。最后得到的结果与图9-5中一样。

268

9.4 进程组

每个进程除了有一个进程ID之外，还属于一个进程组，第10章讨论信号时还会涉及进程组。

进程组是一个或多个进程的集合。通常，它们与同一作业相关联（9.8节详细讨论了作业控制），可以接收来自同一终端的各种信号。每个进程组有一个唯一的进程组ID。进程组ID类似于进程ID——它是一个正整数，并可存放在pid_t数据类型中。函数getpgrp返回调用进程的进程组ID。

```
#include <unistd.h>

pid_t getpgrp(void);
```
返回值：调用进程的进程组ID

在早期BSD派生的系统中，该函数的参数是*pid*，返回该进程的进程组ID。Single UNIX Specification将getpgid函数定义为XSI扩展，它模仿了此种运行行为。

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```
返回值：若成功则返回进程组ID，若出错则返回-1

若*pid*为0，则返回调用进程的进程组ID，于是，

```
getpgid(0);
```

等价于

```
getpgrp();
```

每个进程组都可以有一个组长进程。组长进程的标识是，其进程组ID等于其进程ID。

组长进程可以创建一个进程组，创建该组中的进程，然后终止。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始到其中最后一个进程离开为止的时间区间称为进程组的生存期。进程组中的最后一个进程可以终止，或者转移到另一个进程组。

进程可以通过调用setpgid来加入一个现有的组或者创建一个新进程组（下一节中将说明用setsid也可以创建一个新的进程组）。

269

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```
返回值：若成功则返回0，若出错则返回-1

setpgid函数将*pid*进程的进程组ID设置为*pgid*。如果这两个参数相等，则由*pid*指定的进程变成进程组组长。如果*pid*是0，则使用调用者的进程ID。另外，如果*pgid*是0，则由*pid*指定的进程ID将用作进程组ID。

一个进程只能为它自己或它的子进程设置进程组ID。在它的子进程调用了exec函数之一后，它就不再能改变该子进程的进程组ID。

在大多数作业控制shell中，在fork之后调用此函数，使父进程设置其子进程的进程组ID，并且使子进程设置其自己的进程组ID。这两个调用中有一个是冗余的，但让父子进程都这么做可以保证，在父、子进程认为子进程已进入了该进程组时，这确实已经发生了。如果不这样做

的话，那么fork之后，由于父、子进程运行先后次序的不确定，会造成在一段时间内（父、子进程中只运行了其中一个）子进程组员身份的不确定（取决于哪个进程首先执行），这就产生了竞争条件。

在讨论信号时，将说明如何将一个信号发送给一个进程（由其进程ID标识）或一个进程组（由进程组ID标识）。同样，8.6节的waitpid函数则可用来等待一个进程或者指定进程组中的一个进程终止。

9.5 会话

会话（session）是一个或多个进程组的集合。例如，可以具有图9-6中所示的安排。其中，在一个会话中有三个进程组。

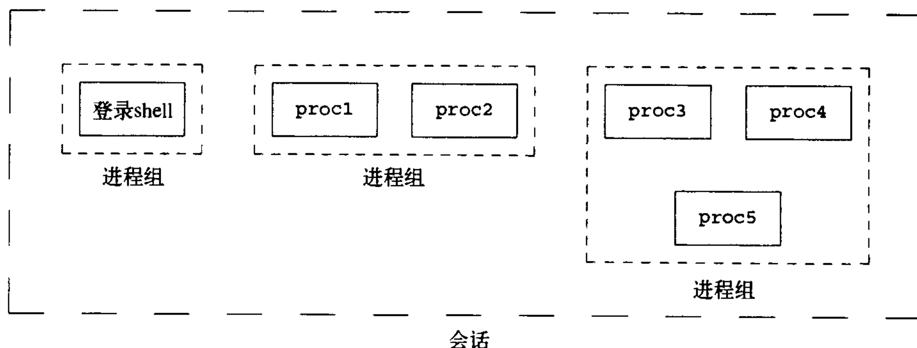


图9-6　进程组和会话中的进程安排

通常是由shell的管道线将几个进程编成一组的。例如，图9-6中的安排可能是由下列形式的shell命令形成的：

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

270

进程调用setsid函数建立一个新会话。

```
#include <unistd.h>

pid_t setsid(void);
```

返回值：若成功则返回进程组ID，若出错则返回-1

如果调用此函数的进程不是一个进程组的组长，则此函数就会创建一个新会话，结果将发生下面3件事：

(1) 该进程变成新*会话首进程*（session leader）。（会话首进程是创建该会话的进程。）此时，该进程是新会话中唯一的进程。

(2) 该进程成为一个新进程组的组长进程。新进程组ID是该调用进程的进程ID。

(3) 该进程没有控制终端（下一节将讨论控制终端）。如果在调用setsid之前该进程有一个控制终端，那么这种联系也会被中断。

如果该调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不会发生这种情况，通常先调用fork，然后使其父进程终止，而子进程则继续。因为子进程继承了父进程的进程组ID，而其进程ID则是新分配的，两者不可能相等，所以这就保证子进程不会是一个进程

组的组长。

Single UNIX Specification只包括“会话首进程”，而没有类似于进程ID和进程组ID的会话ID。显然，会话首进程是具有唯一进程ID的单个进程，所以可以将会话首进程的进程ID视为会话ID。会话ID这一概念是由SVR4引入的。历史上，基于BSD的系统并不支持这个概念，但后来改弦易辙也包括了它。getsid函数返回会话首进程的进程组ID。此函数是Single UNIX Specification的XSI扩展。

> 某些实现（例如Solaris）与Single UNIX Specification保持一致，在实践中避免使用“会话ID”这一短语，作为替代，将其称为“会话首进程的进程组ID”。会话首进程总是一个进程组的组长进程，所以两者是等价的。

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

返回值：若成功则返回会话首进程的进程组ID，若出错则返回−1

如若*pid*是0，getsid返回调用进程的会话首进程的进程组ID。出于安全方面的考虑，某些实现会有如下限制：如若*pid*并不属于调用者所在的会话，那么调用者就不能得到该会话首进程的进程组ID。

271

9.6 控制终端

会话和进程组有一些其他特性：

- 一个会话可以有一个*控制终端*（controlling terminal）。这通常是登录到其上的终端设备（在终端登录情况下）或伪终端设备（在网络登录情况下）。
- 建立与控制终端连接的会话首进程被称为*控制进程*（controlling process）。
- 一个会话中的几个进程组可被分成一个*前台进程组*（foreground process group）以及一个或几个*后台进程组*（background process group）。
- 如果一个会话有一个控制终端，则它有一个前台进程组，会话中的其他进程组则为后台进程组。
- 无论何时键入终端的中断键（常常是DELETE或Ctrl+C），就会将中断信号发送给前台进程组的所有进程。
- 无论何时键入终端的退出键（常常是Ctrl+\），就会将退出信号发送给前台进程组中的所有进程。
- 如果终端接口检测到调制解调器（或网络）已经断开连接，则将挂断信号发送给控制进程（会话首进程）。

272

这些特性示于图9-7中。

通常，我们不必关心控制终端，登录时，将自动建立控制终端。

> POSIX.1将如何分配一个控制终端的机制留由具体实现选择。19.4节中将说明实际步骤。
>
> 当会话首进程打开第一个尚未与一个会话相关联的终端设备时，UNIX系统V派生的系统将此作为控制终端分配给此会话。这假定会话首进程在调用open时没有指定O_NOCTTY标志（见3.3节）。
>
> 当会话首进程用TIOCSCTTY作为*request*参数（第三个参数是空指针）调用ioctl时，基于BSD的

系统为会话分配控制终端。为使此调用成功执行，此会话不能已经有一个控制终端（通常ioctl调用紧跟在setsid调用之后，setsid保证此进程是一个没有控制终端的会话首进程）。除了以兼容模式支持其他系统以外，基于BSD的系统不使用POSIX.1中对open函数所说明的O_NOCTTY标志。

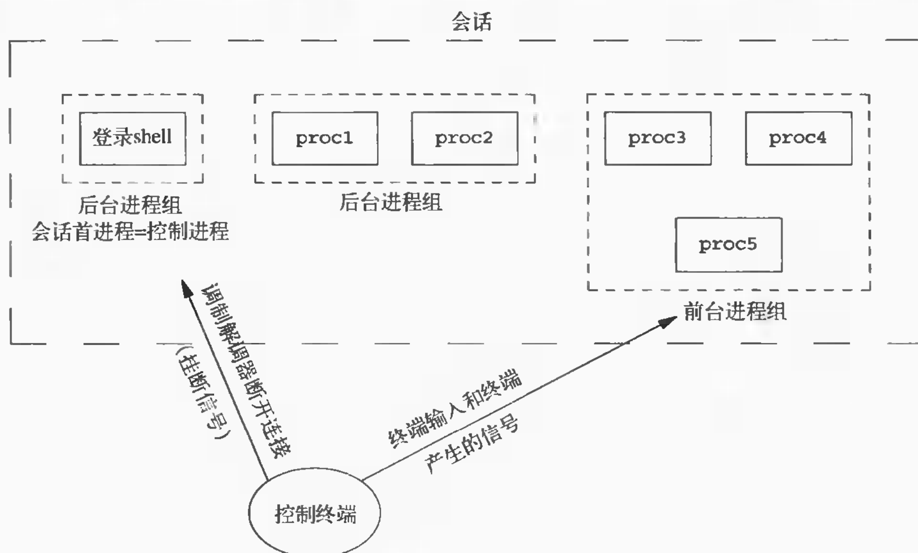


图9-7 显示控制终端的进程组和会话

有时不管标准输入、标准输出是否被重定向，程序都要与控制终端交互。保证程序能读写控制终端的方法是打开文件/dev/tty，在内核中，此特殊文件是控制终端的同义语。自然，如果程序没有控制终端，则打开此设备将失败。

经典的示例是用于读口令的getpass(3)函数（当然，终端回送被关闭）。这一函数由crypt(1)程序调用，而此程序则可用于管道中。例如：

```
crypt < salaries | lpr
```

将文件salaries解密，然后经由管道将输出送至打印假脱机程序。因为crypt从其标准输入读输入文件，所以标准输入不能用于输入口令。但是，crypt的一个设计特征是每次运行此程序时，都应输入加密口令，这样也就阻止了用户将口令存放在文件中（这可能是一个安全性漏洞）。

已经知道有一些方法可以破译crypt程序使用的密码。关于加密文件的详细情况请参见Garfinkel等[2003]。

9.7 tcgetpgrp、tcsetpgrp和tcgetsid函数

需要有一种方法来通知内核哪一个进程组是前台进程组，这样，终端设备驱动程序就能了解将终端输入和终端产生的信号送到何处（见图9-7）。

```
#include <unistd.h>

pid_t tcgetpgrp(int filedes);
                    返回值：若成功则返回前台进程组的进程组ID，若出错则返回-1

int tcsetpgrp(int filedes, pid_t pgrpid);
                    返回值：若成功则返回0，若出错则返回-1
```

函数tcgetpgrp返回前台进程组的进程组ID，该前台进程组与在*filedes*上打开的终端相关联。

如果进程有一个控制终端，则该进程可以调用tcsetpgrp将前台进程组ID设置为*pgrpid*。*pgrpid*的值应当是在同一会话中的一个进程组的ID。*filedes*必须引用该会话的控制终端。

大多数应用程序并不直接调用这两个函数。它们通常由作业控制shell调用。

Single UNIX Specification定义了称为tcgetsid的XSI扩展，给出控制TTY的文件描述符，应用程序就能获得会话首进程的进程组ID。

```
#include <termios.h>

pid_t tcgetsid(int filedes);
```

返回值：若成功则返回会话首进程的进程组ID，若出错则返回-1

需要管理控制终端的应用程序可以调用tcgetsid函数识别出控制终端的会话首进程的会话ID（它等价于会话首进程的进程组ID）。

9.8 作业控制

作业控制是BSD在1980年前后增加的一个特性。它允许在一个终端上启动多个作业（进程组），它控制哪一个作业可以访问该终端，以及哪些作业在后台运行。作业控制要求下面三种形式的支持：

(1) 支持作业控制的shell。

(2) 内核中的终端驱动程序必须支持作业控制。

(3) 内核必须提供对某些作业控制信号的支持。

> SVR3提供了一种不同形式的作业控制，称为shell层（shell layer）。但是POSIX.1选择了BSD形式的作业控制，这也是我们在这里所说明的。在POSIX.1的早期版本中，作业控制支持是可选的，现在则要求所有平台都支持它。

从shell使用作业控制功能角度讲，用户可以在前台或后台启动一个作业。一个作业只是几个进程的集合，通常是一个进程的管道线。例如：

```
vi main.c
```

在前台启动了只有一个进程组成的一个作业。命令

```
pr *.c | lpr &
make all &
```

在后台启动了两个作业。这两个后台作业调用的所有进程都在后台运行。

正如前述，我们需要一个支持作业控制的shell以使用由作业控制提供的功能。对于早期的系统，shell是否支持作业控制比较易于说明。C shell支持作业控制，Bourne shell则不支持，而Korn shell能否支持作业控制则取决于主机是否支持作业控制。但是现在C shell已被移植到并不支持作业控制的系统上（例如系统V的早期版本），而当用名字jsh而不是用sh调用SVR4 Bourne shell时，它支持作业控制。如果主机支持作业控制，则Korn Shell继续支持作业控制。Bourne-again shell也支持作业控制。当各种shell之间的差别并不显著时，我们将只是一般性地说明支持作业控制的shell和不支持作业控制的shell。

274

当启动一个后台作业时，shell赋予它一个作业标识，并打印一个或几个进程ID。下面的脚本显示了Korn shell是如何处理这一点的：

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                           键入回车
[2]  +  Done          pr *.c | lpr &
[1]  +  Done          make all > Make.out &
```

make是作业号1，启动的进程ID是1475。下一个管道线是作业号2，其第一个进程的进程ID是1490。当作业已完成而且我们键入回车时，shell通知我们作业已经完成。键入回车是为了让shell打印其提示符。shell并不在任何随意时刻打印后台作业的状态改变——它只在打印其提示符让用户输入新的命令行之前才这样做。如果不这样处理，则当我们正输入一行时，它也可能输出。

我们可以键入一个影响前台作业的特殊字符——挂起键（一般采用Ctrl+Z）与终端驱动程序进行交互。键入此字符使终端驱动程序将信号SIGTSTP送至前台进程组中的所有进程，后台进程组作业则不受影响。实际上有下面三个特殊字符可使终端驱动程序产生信号，并将它们送至前台进程组：

- 中断字符（一般采用DELETE或Ctrl+C）产生SIGINT。
- 退出字符（一般采用Ctrl+\）产生SIGQUIT。
- 挂起字符（一般采用Ctrl+Z）产生SIGTSTP。

第18章中将说明可将这三个字符更改为用户选择的其他字符，以及如何使终端驱动程序不处理这些特殊字符。

终端驱动程序必须处理与作业控制有关的另一种情况。我们可以有一个前台作业和若干个后台作业，这些作业中哪一个接收我们在终端上键入的字符呢？只有前台作业接收终端输入。如果后台作业试图读终端，那么这并不是一个错误，但是终端驱动程序将检测这种情况，并且向后台作业发送一个特定信号SIGTTIN。该信号通常会暂时停止此后台作业，而shell则向有关用户发出这种情况的通知，然后用户就可用shell命令将此作业转为前台作业运行，于是它就可读终端。下列操作过程演示了这一点：

275

```
$ cat > temp.foo &            在后台启动，但将从标准输入读
[1]     1681
$                             键入回车
[1] + Stopped (SIGTTIN)       cat > temp.foo&
$ fg %1                       使1号作业成为前台作业
cat > temp.foo                shell告诉我们现在哪一个作业在前台
hello, world                  输入1行
^D                            键入文件结束符
$ cat temp.foo                检查该行已送入文件
hello, world
```

shell在后台启动cat进程，但是当cat试图读其标准输入（控制终端）时，终端驱动程序知道它是个后台作业，于是将SIGTTIN信号送至该后台作业。shell检测到其子进程的状态改变（回忆8.6节中对wait和waitpid的讨论），并通知我们该作业已被停止。然后，我们用shell的fg命令将此停止的作业送入前台运行（关于作业控制命令，例如fg和bg的详细情况，以及标识不同作业的各种方法，请参阅有关shell的手册页）。这样做可以使shell将此作业置入前台进程组（tcsetpgrp），并将继续信号（SIGCONT）送给该进程组。因为该作业现在位于前台进程组中，所以它可以读控制终端。

如果后台作业输出到控制终端又将发生什么呢？这是一个我们可以允许或禁止的选项。通常，可以用stty(1)命令改变这一选项（第18章将说明在程序中如何改变这一选项）。下面显示

了这种工作方式：

```
$ cat temp.foo &              在后台执行
[1]     1719
$ hello, world                提示符后出现后台作业的输出
                              键入回车
[1] +  Done          cat temp.foo &
$ stty tostop                 禁止后台作业输出至控制终端
$ cat temp.foo &              在后台再试一次
[1]     1721
$                             键入回车，发现作业已停止
[1] + Stopped(SIGTTOU)          cat temp.foo &
$ fg %1                       在前台恢复停止的作业
cat temp.foo                  shell告诉我们现在哪一个作业在前台
hello, world                  这是该作业的输出
```

在用户禁止后台作业写到控制终端时，cat命令将阻止该作业试图写到其标准输出，因为终端驱动程序将该写操作标识为来自于后台进程，于是向该作业发送SIGTTOU信号。与上面的例子一样，当用户使用shell的fg命令将该作业带至前台时，该作业继续执行直至完成。

图9-8总结了前面已说明的作业控制的某些功能。穿过终端驱动程序框的实线表示：终端I/O和终端产生的信号总是从前台进程组连接到实际终端。对应于SIGTTOU信号的虚线表示；后台进程组进程的输出是否出现在终端是可选的。

276

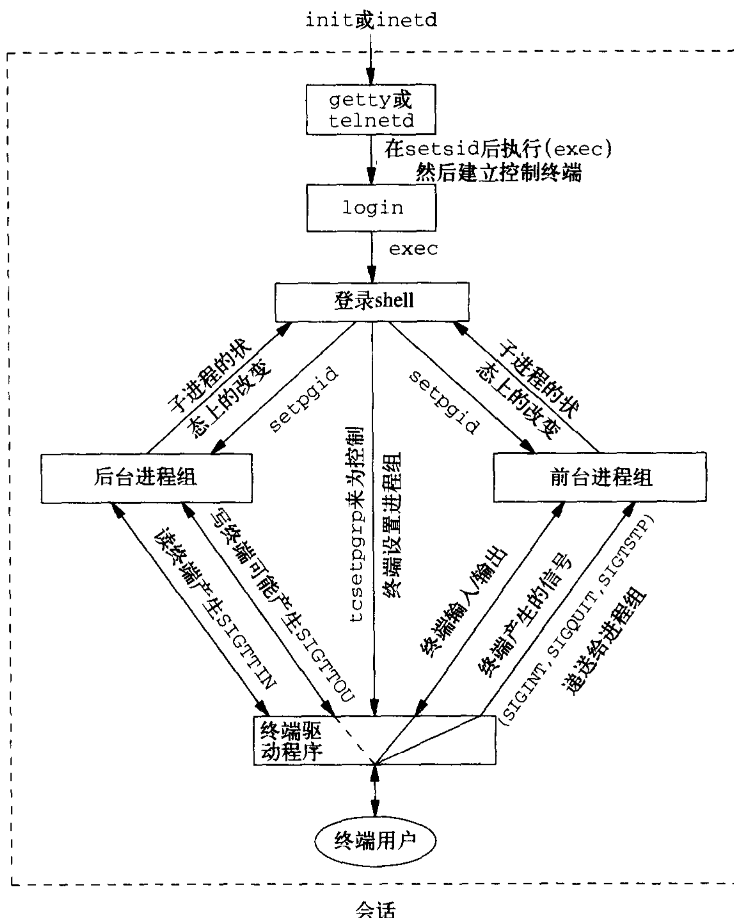


图9-8　具有前台、后台作业以及终端驱动程序的作业控制功能总结

是否需要作业控制是一个有争议的话题。作业控制是在窗口终端广泛得到应用之前设计和实现的。很多人认为设计得好的窗口系统已经免除了对作业控制的需要。某些人抱怨作业控制的实现要求得到内核、终端驱动程序、shell以及某些应用程序的支持，是吃力不讨好的事情。某些人在窗口系统中使用作业控制，他们认为两者都需要。不管你的意见如何，作业控制都是POSIX.1要求的功能。

277

9.9 shell执行程序

让我们研究一下shell是如何执行程序的，以及这与进程组、控制终端和会话等概念的关系。为此，再次使用ps命令。

首先使用不支持作业控制的、在Solaris上运行的经典Bourne shell。如果执行

```
ps -o pid,ppid,pgid,sid,comm
```

则其输出为

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1774   949   949   949 ps
```

ps命令的父进程是shell，这正是我们所期望的。shell和ps命令两者位于同一会话和前台进程组（949）中。因为我们是用一个不支持作业控制的shell执行命令时得到该值的，所以称949为前台进程组。

某些平台支持一个选项，它使ps(1)命令打印与会话控制终端相关联的进程组ID。该值显示在TPGID列中。不幸的是，ps命令的输出在各个UNIX版本之间有所不同。例如，Solaris 9不支持该选项。在FreeBSD 5.2.1和Mac OS X 10.3中，命令

```
ps -o pid,ppid,pgid,sess,tpgid,command
```

在Linux 2.4.22中，命令

```
ps -o pid,ppid,pgrp,session,tpgid,comm
```

都会准确打印我们想要的信息。

注意，将进程与终端进程组ID（TPGID列）关联起来有点用词不当。进程并没有终端进程控制组。进程属于一个进程组，而进程组属于一个会话。会话可能有也可能没有控制终端。如果它确实有一个控制终端，则此终端设备知道其前台进程的进程组ID。这个值可以用tcsetpgrp函数在终端驱动程序中设置（见图9-8）。前台进程组ID是终端的一个属性，而不是进程的属性。取自终端设备驱动程序的该值是ps在TPGID列中打印的值。如果ps发现此会话没有控制终端，则它在该列打印-1。

如果在后台执行该命令：

```
ps -o pid,ppid,pgid,sid,comm &
```

278

则唯一改变的值是命令的进程ID。

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1812   949   949   949 ps
```

因为这种shell不知道作业控制，所以后台作业没有构成另一个进程组，也没有从后台作业处取走控制终端。

现在看一看Bourne shell如何处理管道线。执行命令

```
ps -o pid,ppid,pgid,sid,comm | cat1
```

时其输出是

```
 PID  PPID  PGID   SID COMMAND
 949   947   949   949 sh
1823   949   949   949 cat1
1824  1823   949   949 ps
```

(程序cat1只是标准cat程序的一个副本，但名字不同。本节还将使用cat的另一个名为cat2的副本。在一个管道线中使用两个cat副本时，不同的名字可使我们将它们区分开来。）注意，管道线中的最后一个进程是shell的子进程，该管道线中的第一个进程则是最后一个进程的子进程。从中可以看出，shell fork一个它自身的副本，然后此副本再为管道线中的每条命令各fork一个进程。

如果在后台执行此管道线：

```
ps -o pid,ppid,pgid,sid,comm | cat1 &
```

则只有进程ID改变了。因为shell并不处理作业控制，后台进程的进程组ID仍是949，如同会话的进程组ID一样。

如果一个后台进程试图读其控制终端，则会发生什么呢？例如，若执行：

```
cat > temp.foo &
```

在有作业控制时，后台作业被放在后台进程组中，如果后台作业试图读控制终端，则会产生信号SIGTTIN。在没有作业控制时，其处理方法是：如果该进程自己没有重定向标准输入，则shell自动将后台进程的标准输入重定向到/dev/null。读/dev/null则产生一个文件结束。这就意味着后台cat进程立即读到文件尾，并正常结束。

上面说明了对后台进程通过其标准输入访问控制终端的适当的处理方法，但是，如果一个后台进程打开/dev/tty并且读该控制终端，又将怎样呢？对此问题的回答是“看情况”。但是这很可能不是我们所期望的。例如：

```
crypt < salaries | lpr &
```

279 就是这样一条管道线。我们在后台运行它，但是crypt程序打开/dev/tty，更改终端的特性（禁止回送），然后读该设备，最后重置该终端特性。当执行这条后台管道线时，crypt在终端上打印提示符“Password:”，但是shell读取了我们所输入的加密口令，并试图执行以加密口令为名称的命令。我们输送给shell的下一行则被crypt进程取为口令行，于是不能正确对文件加密，结果将一堆无用信息送到了打印机。在这里，我们有了两个进程，它们试图同时读同一设备，其结果则依赖于系统。前面说明的作业控制以较好的方式处理一个终端在多个进程间的复用。

返回到Bourne shell实例，如果在一条管道线中执行三个进程，我们就可以检验Bourne shell使用的进程控制方式：

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

其输出为

```
 PID  PPID  PGID   SID COMMAND
 949   947   949   949 sh
1988   949   949   949 cat2
1989  1988   949   949 ps
1990  1988   949   949 cat1
```

如果在你的系统上，输出的命令名不正确，那也不必为此感到惊慌。有时你可能会得类似如下的输出：

```
 PID  PPID  PGID   SID COMMAND
 949   947   949   949 sh
1831   949   949   949 sh
```

```
 1832  1831   949   949 ps
 1833  1831   949   949 sh
```

造成此种结果的原因是，ps进程与shell产生竞争条件，shell创建一个子进程并由它执行cat命令。在这种情况下，当ps已经获得进程列表并打印时，shell尚未完成exec调用。

再重申一遍，该管道线中的最后一个进程是shell的子进程，而执行管道线中其他命令的进程则是该最后一个进程的子进程。图9-9显示了所发生的情况。因为该管道线中的最后一个进程是登录shell的子进程，当该进程（cat2）终止时，shell得到通知。

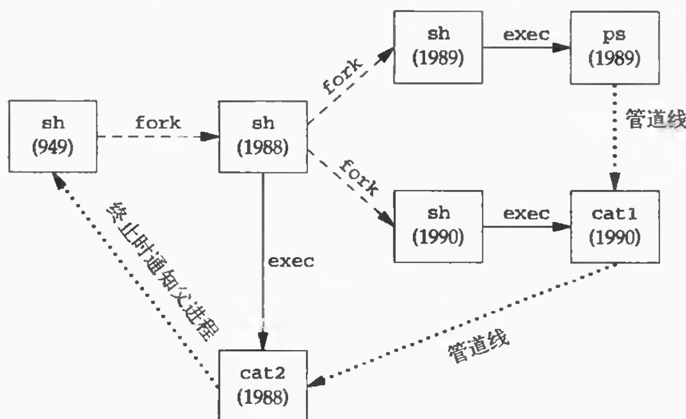


图9-9 Bourne shell调用管道线ps | cat1 | cat2时的进程

现在让我们用一个运行在Linux上的作业控制shell来检验同一个例子。这将显示这些shell处理后台作业的方法。在本例中将使用Bourne-again shell，用其他作业控制shell得到的结果几乎完全一样。

```
ps -o pid,ppid,pgrp,session,tpgid,comm
```

其输出为

```
  PID  PPID  PGRP  SESS TPGID COMMAND
 2837  2818  2837  2837  5796 bash
 5796  2837  5796  2837  5796 ps
```

（从本例开始，以粗体显示前台进程组。）我们立即看到了与Bourne shell例子的区别。Bourne-again shell将前台作业（ps）放入它自己的进程组（5796）中。ps命令是组长进程，并是该进程组中的唯一进程。

280

进一步讲，此进程组具有控制终端，所以它是前台进程组。我们的登录shell在执行ps命令时是后台进程组。但需要注意的是，这两个进程组2837和5796都是同一会话的成员。事实上，在本节的各实例中会话决不会改变。

在后台执行此进程：

```
ps -o pid,ppid,pgrp,session,tpgid,comm &
```

其输出为

```
  PID  PPID  PGRP  SESS TPGID COMMAND
 2837  2818  2837  2837  2837 bash
 5797  2837  5797  2837  2837 ps
```

再一次，ps命令被放入它自己的进程组中，但是此时进程组（5797）不再是前台进程组，而是一个后台进程组。TPGID 2837指示前台进程组是用户的登录shell。

按下列方式在一个管道线中执行两个进程：

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1
```

其输出为

```
 PID  PPID  PGRP  SESS TPGID COMMAND
2837  2818  2837  2837  5799 bash
5799  2837  5799  2837  5799 ps
5800  2837  5799  2837  5799 cat1
```

两个进程ps和cat1都在一个新进程组（5799）中，这是一个前台进程组。在本例和类似的Bourne shell实例之间能看到另一个区别。Bourne shell首先创建将执行管道线中最后一条命令的进程，而此进程是第一个进程的父进程。在这里，Bourne-again Shell是两个进程的父进程。但是，如果在后台执行此管道线：

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1 &
```

其结果是类似的，但是ps和cat1现在都处于同一后台进程组中。

```
 PID  PPID  PGRP  SESS TPGID COMMAND
2837  2818  2837  2837  2837 bash
5801  2837  5801  2837  2837 ps
5802  2837  5801  2837  2837 cat1
```

注意，如果使用的shell不同，那么它创建各个进程的顺序也可能不同。

9.10 孤儿进程组

我们曾提及，一个其父进程已终止的进程称为孤儿进程（orphan process），这种进程由init进程“收养”。现在我们要说明整个进程组也可成为“孤儿”，以及POSIX.1如何处理它。

考虑一个进程，它fork了一个子进程然后终止。这在系统中是经常发生的，并无异常之处，但是在父进程终止时，如果该子进程停止（用作业控制），则会发生什么情况呢？子进程如何继续，以及子进程是否知道它已经是孤儿进程？图9-10显示了这种情形：父进程已经fork了子进程，该子进程停止，父进程则将退出。

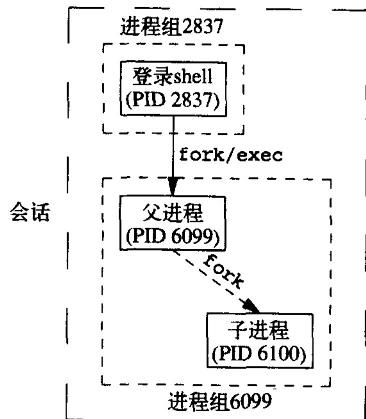


图9-10 将要成为孤儿的进程组实例

产生这种情形的程序示于程序清单9-1中。下面要说明该程序的某些新特征。这里，假定使用了一个作业控制shell。回忆前面所述，shell将前台进程放在它（指前台进程）自已的进程组（本例中是6099）中，shell则留在自己的进程组（2837）内。子进程继承其父进程（6099）的进程组。在fork之后：

- 父进程休眠5秒钟，这是一种让子进程在父进程终止之前运行的一种权宜之计。
- 子进程为挂断信号（SIGHUP）建立信号处理程序。这样就能观察到SIGHUP信号是否已发送到子进程。（第10章将讨论信号处理程序。）
- 子进程用kill函数向其自身发送停止信号（SIGTSTP）。这停止了子进程，类似于用终端挂起字符（Ctrl+Z）停止一个前台作业。

- 当父进程终止时，该子进程成为孤儿进程，所以其父进程ID成为1，也就是init进程ID。
- 现在，子进程成为一个孤儿进程组的成员。POSIX.1将**孤儿进程组**（orphaned process group）定义为：该组中每个成员的父进程要么是该组的一个成员，要么不是该组所属会话的成员。对孤儿进程组的另一种描述如下：一个进程组不是孤儿进程组的条件是，该组中有一个进程，其父进程在属于同一会话的另一个组中。如果进程组不是孤儿进程组，那么在属于同一会话的另一个组中的父进程就有机会重新启动该组中停止的进程。在这里，进程组中每一个进程的父进程（例如，进程6100的父进程是进程1）都属于另一个会话。所以此进程组是孤儿进程组。
- 因为在父进程终止后，进程组成为孤儿进程组，POSIX.1要求向新的孤儿进程组中处于停止状态的每一个进程发送挂断信号（SIGHUP），接着又向其发送继续信号（SIGCONT）。
- 在处理了挂断信号后，子进程继续。对挂断信号的系统默认动作是终止该进程，为此必须提供一个信号处理程序以捕捉该信号。因此，我们期望sig_hup函数中的printf会在pr_ids函数中的printf之前执行。

程序清单9-1 创建一个孤儿进程组

```
#include "apue.h"
#include <errno.h>

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %d\n", getpid());
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %d, ppid = %d, pgrp = %d, tpgrp = %d\n",
        name, getpid(), getppid(), getpgrp(), tcgetpgrp(STDIN_FILENO));
    fflush(stdout);
}

int
main(void)
{
    char    c;
    pid_t   pid;

    pr_ids("parent");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        sleep(5);       /* sleep to let child stop itself */
        exit(0);        /* then parent exits */
    } else {            /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);    /* establish signal handler */
        kill(getpid(), SIGTSTP);    /* stop ourself */
        pr_ids("child");    /* prints only if we're continued */
        if (read(STDIN_FILENO, &c, 1) != 1)
            printf("read error from controlling TTY, errno = %d\n",
                errno);
        exit(0);
    }
}
```

下面是程序清单9-1的输出：

```
$ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error from controlling TTY, errno = 5
```

注意，因为两个进程即登录shell和子进程都写向终端，所以shell提示符和子进程的输出一起出现。正如我们所期望的那样，子进程的父进程ID变成1。

在子进程中调用pr_ids后，程序试图读标准输入。正如前述，当后台进程组试图读控制终端时，则对该后台进程组产生SIGTTIN。但在这里，这是一个孤儿进程组，如果内核用此信号停止它，则此进程组中的进程就再也不会继续。POSIX.1规定，在这种情形下read返回出错，并将其errno设置为EIO（在作者所用的系统中其值是5）。

283～284

最后，要注意的是父进程终止时，子进程被置入后台进程组中，因为父进程是由shell作为前台作业执行的。□

在19.5节的pty程序中将会看到孤儿进程组的另一个例子。

9.11 FreeBSD实现

上面说明了进程、进程组、会话和控制终端的各种属性，值得观察一下所有这些是如何实现的。下面简要说明FreeBSD的实现。SVR4实现的某些详细情况则请参阅Williams[1989]。图9-11显示了FreeBSD使用的各种有关数据结构。

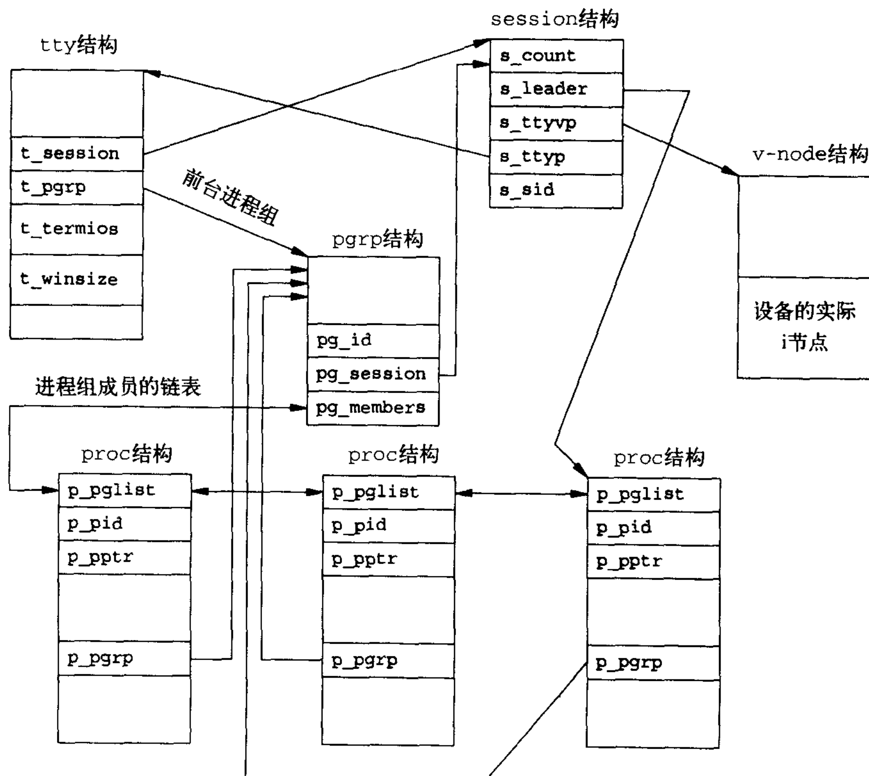


图9-11　会话和进程组的FreeBSD实现

285

下面说明图9-11中标出的各个字段。从session结构开始。每个会话都分配一个session结构（例如，每次调用setsid时）。

- s_count是该会话中的进程组数。当此计数器减至0时，则可释放此结构。
- s_leader是指向会话首进程proc结构的指针。
- s_ttyvp是指向控制终端vnode结构的指针。
- s_ttyp是指向控制终端tty结构的指针。
- s_sid是会话ID。请记住会话ID这一概念并非Single UNIX Specification的组成部分。

在调用setsid时，在内核中分配一个新的会话结构。现在s_count设置为1，s_leader设置为调用进程proc结构的指针，s_sid设置为进程ID，因为新会话没有控制终端，所以s_ttyvp和s_ttyp设置为空指针。

接着说明tty结构。每个终端设备和每个伪终端设备均在内核中分配这样一种结构（第19章将对伪终端作更多说明）。

- t_session指向将此终端作为控制终端的session结构（注意，tty结构指向session结构，session结构也指向tty结构）。终端在失去载波信号时使用此指针将挂断信号送给会话首进程（见图9-7）。
- t_pgrp指向前台进程组的pgrp结构。终端驱动程序用此字段将信号送至前台进程组。由输入特殊字符（中断、退出和挂起）而产生的三个信号被送至前台进程组。
- t_termios是包含所有这些特殊字符以及与该终端有关信息（例如，波特率、回送打开或关闭等）的结构。第18章将回过头来说明此结构。
- t_winsize是包含终端窗口当前尺寸的winsize结构。当终端窗口尺寸改变时，信号SIGWINCH被送至前台进程组。18.12节将说明如何设置和获取终端的当前窗口尺寸。

注意，为了找到特定会话的前台进程组，内核不得不从session结构开始，然后用s_ttyp得到控制终端的tty结构，接着用t_pgrp得到前台进程组的pgrp结构。pgrp结构包含一特定进程组的信息。其中各相关字段是：

- pg_id是进程组ID。
- pg_session指向此进程组所属会话的session结构。
- pg_members是指向作为此进程组成员的proc结构列表的指针。proc结构中的p_pglist结构是一个双向链表项，它同时指向此组中的下一个进程和前一个进程，依此类推，直到进程组中最后一个进程的proc结构中遇到一个空指针。

286

proc结构包含一个进程的所有信息。

- p_pid包含进程ID。
- p_pptr是指向父进程proc结构的指针。
- p_pgrp指向本进程所属进程组的pgrp结构的指针。
- 如前所述，p_pglist是一个结构，其中包含两个指针，分别指向进程组中的前一个进程和下一个进程。

最后还有一个vnode结构。在打开控制终端设备时分配此结构。进程对/dev/tty的所有引用都通过vnode结构。图9-11中显示实际i节点是v节点的一部分。

9.12 小结

本章说明了进程组之间的关系——会话，它由若干个进程组组成。作业控制是当今很多

UNIX系统所支持的功能，本章说明了它是如何由支持作业控制的shell实现的。在这些进程关系中也涉及到了进程的控制终端/dev/tty。

所有这些进程的关系都使用了很多信号方面的功能。下一章将详细讨论UNIX中的信号机制。

习题

9.1 考虑6.8节中说明的utmp和wtmp文件，为什么注销记录是由init进程编写的？对于网络登录的处理与此相同吗？

9.2 编写一小段程序，要求调用fork并使子进程建立一个新的会话。验证子进程变成了进程组组长且不再具有控制终端。

287

第10章

信　　号

10.1 引言

信号是软件中断。很多比较重要的应用程序都需处理信号。信号提供了一种处理异步事件的方法，例如，终端用户键入中断键，则会通过信号机制停止一个程序，或及早终止管道中的下一个程序。

UNIX的早期版本就已经提供了信号机制，但是这些系统（例如V7）所提供的信号模型并不可靠。信号可能丢失，而且在执行临界区代码时，进程很难关闭所选择的信号。4.3BSD和SVR3对信号模型都做了更改，增加了可靠信号机制。但是Berkeley和AT&T所做的更改之间并不兼容。幸运的是POSIX.1对可靠信号例程进行了标准化，这正是本章所要说明的。

本章先对信号机制进行综述，并说明每种信号的一般用法。然后分析早期实现的问题。在分析存在的问题之后再说明解决这些问题的方法，这种安排有助于加深对改进机制的理解。本章也包含了很多并非完全正确的实例，这样做的目的是为了对其不足之处进行讨论。

10.2 信号概念

首先，每个信号都有一个名字。这些名字都以三个字符SIG开头。例如，SIGABRT是夭折信号，当进程调用abort函数时产生这种信号。SIGALRM是闹钟信号，当由alarm函数设置的计时器超时后产生此信号。V7有15种不同的信号，SVR4和4.4BSD均有31种不同的信号。FreeBSD 5.2.1、Mac OS X 10.3以及Linux 2.4.22支持31种不同的信号，而Solaris 9则支持38种不同的信号。另外，Linux和Solaris都支持应用程序额外定义的信号，将其作为实时扩展（本书不包括POSIX实时扩展，有关信息请参阅Gallmeister[1995]）。

在头文件<signal.h>中，这些信号都被定义为正整数（信号编号）。

> 实际上，实现将各信号定义在另一个头文件中，但是该头文件又包括在<signal.h>中。内核包括对用户级应用程序有意义的头文件，这被认为是一种糟糕的形式，所以如若应用程序和内核两者都需使用同一定义，那么就将有关信息放置在内核头文件中，然后用户级头文件再包括该内核头文件。于是，FreeBSD 5.2.1和Mac OS X 10.3将信号定义在<sys/signal.h>中，Linux 2.4.22将信号定义在<bits/signum.h>中，Solaris 9则将信号定义在<sys/iso/signal_iso.h>中。

不存在编号为0的信号。在10.9节中将会看到，kill函数对信号编号0有特殊的应用。POSIX.1将此种信号编号值称为空信号。

很多条件可以产生信号：

- 当用户按某些终端键时，引发终端产生的信号。在终端上按DELETE键（或者很多系统

中的Ctrl+C键）通常产生中断信号（SIGINT）。这是停止一个已失去控制的程序的方法。（第18章将说明此信号可被映射为终端上的任一字符。）

- 硬件异常产生信号：除数为0、无效的内存引用等等。这些条件通常由硬件检测到，并将其通知内核。然后内核为该条件发生时正在运行的进程产生适当的信号。例如，对执行一个无效内存引用的进程产生SIGSEGV信号。
- 进程调用kill(2)函数可将信号发送给另一个进程或进程组。自然，对此有所限制：接收信号进程和发送信号进程的所有者必须相同，或者发送信号进程的所有者必须是超级用户。
- 用户可用kill(1)命令将信号发送给其他进程。此命令只是kill函数的接口。常用此命令终止一个失控的后台进程。
- 当检测到某种软件条件已经发生，并应将其通知有关进程时也产生信号。这里指的不是硬件产生的条件（如除以0），而是软件条件。例如SIGURG（在网络连接上传来带外数据时产生）、SIGPIPE（在管道的读进程已终止后，一个进程写此管道时产生），以及SIGALRM（进程所设置的闹钟时钟超时时产生）。

信号是异步事件的经典实例。产生信号的事件对进程而言是随机出现的。进程不能简单地测试一个变量（例如errno）来判别是否出现了一个信号，而是必须告诉内核“在此信号出现时，请执行下列操作”。

290

可以要求内核在某个信号出现时按照下列三种方式之一进行处理，我们称之为信号的处理或者与信号相关的动作。

(1) 忽略此信号。大多数信号都可使用这种方式进行处理，但有两种信号却决不能被忽略。它们是SIGKILL和SIGSTOP。这两种信号不能被忽略的原因是：它们向超级用户提供了使进程终止或停止的可靠方法。另外，如果忽略某些由硬件异常产生的信号（例如非法内存引用或除以0），则进程的运行行为是未定义的。

(2) 捕捉信号。为了做到这一点，要通知内核在某种信号发生时调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理。例如，若正在运行一个命令解释器，它将用户的输入解释为命令并执行之，当用户用键盘产生中断信号时，很可能希望该命令解释器返回到主循环，终止正在为该用户执行的命令。如果捕捉到SIGCHLD信号，则表示一个子进程已经终止，所以此信号的捕捉函数可以调用waitpid以取得该子进程的进程ID以及它的终止状态。又例如，如果进程创建了临时文件，那么可能要为SIGTERM信号编写一个信号捕捉函数以清除临时文件（SIGTERM是终止信号，kill命令传送的系统默认信号是终止信号）。注意，不能捕捉SIGKILL和SIGSTOP信号。

(3) 执行系统默认动作。表10-1给出了针对每一种信号的系统默认动作。注意，针对大多数信号的系统默认动作是终止进程。

表10-1列出了所有信号的名字，说明了哪些系统支持此信号以及针对这些信号的系统默认动作。在SUS列中，• 表示此种信号被定义为基本POSIX.1规范部分，“XSI”表示该信号定义为该基本规范的XSI扩展。

在“默认动作”列中，“终止+core”表示在进程当前工作目录的core文件中复制该进程的存储映像（该文件名为core，由此可以看出这种功能很久以前就是UNIX的一部分）。大多数UNIX调试程序都使用core文件以检查进程终止时的状态。

表10-1 UNIX系统信号

名字	说明	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	默认动作
SIGABRT	异常终止 (abort)	•	•	•	•	•	•	终止+core
SIGALRM	超时 (alarm)		•	•	•	•	•	终止
SIGBUS	硬件故障		•	•	•	•	•	终止+core
SIGCANCEL	线程库内部使用						•	忽略
SIGCHLD	子进程状态改变		•	•	•	•	•	忽略
SIGCONT	使暂停进程继续		•	•	•	•	•	继续/忽略
SIGEMT	硬件故障			•	•	•	•	终止+core
SIGFPE	算术异常	•	•	•	•	•	•	终止+core
SIGFREEZE	检查点冻结						•	忽略
SIGHUP	连接断开		•	•	•	•	•	终止
SIGILL	非法硬件指令	•	•	•	•	•	•	终止+core
SIGINFO	键盘状态请求			•		•		忽略
SIGINT	终端中断符	•	•	•	•	•	•	终止
SIGIO	异步I/O			•	•	•	•	终止/忽略
SIGIOT	硬件故障			•	•	•	•	终止+core
SIGKILL	终止		•	•	•	•	•	终止
SIGLWP	线程库内部使用						•	忽略
SIGPIPE	写至无读进程的管道		•	•	•	•	•	终止
SIGPOLL	可轮询事件 (poll)		XSI		•		•	终止
SIGPROF	梗概时间超时 (setitimer)		XSI	•	•	•	•	终止
SIGPWR	电源失效/重启动				•		•	终止/忽略
SIGQUIT	终端退出符		•	•	•	•	•	终止+core
SIGSEGV	无效内存引用	•	•	•	•	•	•	终止+core
SIGSTKFLT	协处理器栈故障				•			终止
SIGSTOP	停止		•	•	•	•	•	暂停进程
SIGSYS	无效系统调用		XSI	•	•	•	•	终止+core
SIGTERM	终止	•	•	•	•	•	•	终止
SIGTHAW	检查点解冻						•	忽略
SIGTRAP	硬件故障		XSI	•	•	•	•	终止+core
SIGTSTP	终端停止符		•	•	•	•	•	暂停进程
SIGTTIN	后台读控制tty		•	•	•	•	•	暂停进程
SIGTTOU	后台写至控制tty		•	•	•	•	•	暂停进程
SIGURG	紧急情况 (套接字)		•	•	•	•	•	忽略
SIGUSR1	用户定义的信号		•	•	•	•	•	终止
SIGUSR2	用户定义的信号		•	•	•	•	•	终止
SIGVTALRM	虚拟时间闹钟 (setitimer)		XSI	•	•	•	•	终止
SIGWAITING	线程库内部使用						•	忽略
SIGWINCH	终端窗口大小改变			•	•	•	•	忽略
SIGXCPU	超过CPU限制 (setrlimit)		XSI	•	•	•	•	终止+core/忽略
SIGXFSZ	超过文件长度限制 (setrlimit)		XSI	•	•	•	•	终止+core/忽略
SIGXRES	超过资源控制						•	忽略

core文件是大多数UNIX系统的实现特征。虽然该特征不是POSIX.1的组成部分，但曾经提及这可能作为实现特定动作成为Single UNIX Specification的XSI扩展。

在不同的实现中，core文件的名字可能不同。例如，在FreeBSD 5.2.1中，core文件名为*cmdname*.core，其中*cmdname*是接收到信号的进程所执行的命令名。在Mac OS X 10.3中，core文件名是core.*pid*，其中，*pid*是接收到信号的进程的ID。（这些系统允许经sysctl参数配置core文件名。）

大多数实现在相应进程的当前工作目录中存放core文件；但Mac OS X将所有core文件都放置在/cores目录中。

291～292

在下列条件下不产生core文件：(a) 进程是设置用户ID的，而且当前用户并非程序文件的所有者，(b) 进程是设置组ID的，而且当前用户并非该程序文件的组所有者，(c) 用户没有写当前工作目录的权限，(d) 文件已存在，而且用户对该文件设有写权限，(e) 文件太大（回忆7.11节中的RLIMIT_CORE限制）。core文件的权限（假定该文件在此之前并不存在）通常是用户读/写，但Mac OS X只设置为用户读。

在表10-1说明中的“硬件故障”对应于实现定义的硬件故障。这些名字中有很多取自UNIX早先在PDP-11上的实现。请查看你所使用的系统手册，以确切地弄清楚这些信号对应于哪些错误类型。

下面较详细地逐一说明这些信号。

SIGABRT 调用abort函数时（见10.17节）产生此信号。进程异常终止。

SIGALRM 在用alarm函数设置的计时器超时时，产生此信号（详细情况见10.10节）若由setitimer(2)函数设置的间隔时间超时时，也会产生此信号。

SIGBUS 指示一个实现定义的硬件故障。当出现某些类型的内存故障时（如14.9节中说明的），实现常常产生此种信号。

SIGCANCEL 这是Solaris线程库内部使用的信号。它不供一般应用。

SIGCHLD 在一个进程终止或停止时，将SIGCHLD信号发送给其父进程。按系统默认，将忽略此信号。如果父进程希望被告知其子进程的这种状态改变，则应捕捉此信号。信号捕捉函数中通常要调用一种wait函数以取得子进程ID和其终止状态。

系统V的早期版本有一个名为SIGCLD（无H）的类似信号。这一信号具有与其他信号不同的语义，SVR2的手册页警告在新的程序中尽量不要使用这种信号。令人惊讶的是在SVR3和SVR4版本的手册页中，该警告消失了。应用程序应当使用标准的SIGCHLD信号，但应了解，为了向后兼容，很多系统定义了与SIGCHLD等同的SIGCLD。（如果有使用SIGCLD的软件，需要查阅系统手册，了解它的具体语义。）10.7节将讨论这两个信号。

SIGCONT 此作业控制信号被发送给需要继续运行，但当前处于停止状态的进程。如果接收到此信号的进程处于停止状态，则系统默认动作是使该进程继续运行，否则默认动作是忽略此信号。例如，全屏幕编辑器在捕捉到此信号后，使用信号处理程序发出重新绘制终端屏幕的通知。关于进一步的情况见10.20节。

SIGEMT 指示一个实现定义的硬件故障。

EMT这一名字来自PDP-11的“仿真器陷入”（emulator trap）指令。并非所有平台都支持此信号。例如，Linux只对SPARC、MIPS和PA_RISC等体系结构支持SIGEMT。

293

SIGFPE　此信号表示一个算术运算异常，例如除以0，浮点溢出等。

SIGFREEZE　此信号仅由Solaris定义。它用于通知进程在冻结系统状态之前需要采取特定动作，例如当系统进入冬眠或挂起模式时可能需要执行这种处理。

SIGHUP　如果终端接口检测到一个连接断开，则将此信号发送给与该终端相关的控制进程（会话首进程）。见图9-11，此信号被送给session结构中的s_leader字段所指向的进程。仅当终端的CLOCAL标志没有设置时，在上述条件下才产生此信号。（如果所连接的终端是本地的，则设置该终端的CLOCAL标志。它告诉终端驱动程序忽略所有调制解调器的状态行。第18章将说明如何设置此标志。）

注意，接到此信号的会话首进程可能在后台，例如，参见图9-7。这有别于由终端正常产生的几个信号（中断、退出和挂起），这些信号总是传递给前台进程组。

如果会话首进程终止，则也产生此信号。在这种情况下，此信号将被发送给前台进程组中的每一个进程。

通常用此信号通知守护进程（见第13章），以重新读取它们的配置文件。为此目的选用SIGHUP的理由是，守护进程不会有控制终端，而且通常决不会接收到这种信号。

SIGILL　此信号指示进程已执行一条非法硬件指令。

原先，4.3BSD的abort函数产生此信号。现在该函数产生SIGABRT信号。

SIGINFO　这是一种BSD信号，当用户按状态键（一般采用Ctrl+T）时，终端驱动程序产生此信号并送至前台进程组中的每一个进程（见图9-8）。此信号通常导致在终端上显示前台进程组中各进程的状态信息。

除了在Alpha平台上将SIGINFO定义为与SIGPWR具有相同的值之外，Linux不支持这种信号。

SIGINT　当用户按中断键（一般采用DELETE或Ctrl+C）时，终端驱动程序产生此信号并送至前台进程组中的每一个进程（见图9-8）。当一个进程在运行时失控，特别是它正在屏幕上产生大量不需要的输出时，常用此信号终止它。

SIGIO　此信号指示一个异步I/O事件。在14.6.2节中将对此进行讨论。

294

在表10-1中，针对SIGIO的系统默认动作是终止或忽略。不幸的是，这依赖于系统。在系统V中，SIGIO与SIGPOLL相同，因此其默认动作是终止此进程。在BSD中，其默认动作是忽略此信号。

Linux 2.4.22和Solaris 9将SIGIO定义为与SIGPOLL具有相同的值，所以默认行为是终止该进程。在FreeBSD 5.2.1和Mac OS X 10.3中，默认行为是忽略该信号。

SIGIOT　这指示一个实现定义的硬件故障。

IOT这个名字来自于PDP-11，它是PDP-11计算机“输入/输出TRAP”（input/output TRAP）指令的缩写。在系统V的早期版本中，由abort函数产生此信号。该函数现在产生SIGABRT信号。

在FreeBSD 5.2.1、Linux 2.4.22、Mac OS X 10.3和Solaris 9中将SIGIOT定义为与SIGABRT具有相同的值。

SIGKILL　这是两个不能被捕捉或忽略的信号之一。它向系统管理员提供了一种可以杀死任一进程的可靠方法。

SIGLWP　此信号由Solaris线程库内部使用，并不作一般使用。

SIGPIPE　如果在写到管道时读进程已终止，则产生此信号。15.2节将说明管道。当类型为SOCK_STREAM的套接字已不再连接时，进程写到该套接字也产生此信号。我们将在第16章说明套接字。

SIGPOLL　当在一个可轮询设备上发生一特定事件时产生此信号。14.5.2节将说明poll函数和此信号。它起源于SVR3，并松散对应于BSD的SIGIO和SIGURG信号。

在Linux和Solaris中，SIGPOLL被定义为与SIGIO具有相同的值。

SIGPROF　当setitimer(2)函数设置的梗概统计间隔计时器（profiling interval timer）已到期时产生此信号。

SIGPWR　这是一种依赖于系统的信号。它主要用于具有不间断电源（UPS）的系统。如果电源失效，则UPS起作用，而且通常软件会接到通知。在这种情况下，系统依靠蓄电池电源继续运行，所以无须任何处理。但是如果蓄电池也将不能支持工作，则软件通常会再次接到通知，此时，系统必须在15～30秒内使其各部分都停止运行。此时应当发送SIGPWR信号。在大多数系统中，接到蓄电池电压过低信息的进程将信号SIGPWR发送给init进程，然后由init处理停机操作。

Linux 2.4.22和Solaris 9在inittab文件中有两个项用于此种目的，它们是powerfail以及powerwait（或powerokwait）。

在表10-1中，我们将SIGPWR的默认动作标记为："终止"或者"忽略"。不幸的是，这种默认依赖于系统。Linux的默认动作是终止相关进程，Solaris的默认动作是忽略该信号。

295

SIGQUIT　当用户在终端上按退出键（一般采用Ctrl+\）时，产生此信号，并送至前台进程组中的所有进程（见图9-8）。此信号不仅会终止前台进程组（如SIGINT所做的那样），同时还会产生一个core文件。

SIGSEGV　该信号指示进程进行了一次无效内存引用。

名字SEGV表示"段违例（segmentation violation）"。

SIGSTKFLT　此信号仅由Linux定义。它出现在Linux的早期版本，旨在用于数学协处理器的栈故障。该信号并非由内核产生，但仍保留以向后兼容。

SIGSTOP　这是一个作业控制信号，用于停止一个进程。它类似于交互停止信号（SIGTSTP），但是SIGSTOP不能被捕捉或忽略。

SIGSYS　该信号指示一个无效的系统调用。由于某种未知的原因，进程执行了一条机器指令，内核认为这是一个系统调用，但该指令指示系统调用类型的参数却是无效的。这种情况是可能发生的，例如，若用户编写了一道使用新

系统调用的程序，然后尝试运行该程序的二进制可执行代码，而所用的操作系统却是不支持该系统调用的较早版本，于是就会出现上述情况。

SIGTERM 这是由kill(1)命令发送的系统默认终止信号。

SIGTHAW 此信号仅由Solaris定义。当系统恢复运行被挂起的操作时，该信号用于通知相关进程，它们需要采取特殊的动作。

SIGTRAP 指示一个实现定义的硬件故障。

此信号名来自于PDP-11的TRAP指令。当执行断点指令时，实现常用此信号将控制转移至调试程序。

SIGTSTP 交互式停止信号，当用户在终端上按挂起键（一般采用Ctrl+Z）时，终端驱动程序产生此信号。该信号送至前台进程组中的所有进程（参见图9-8）。

不幸的是，停止（stop）这个术语具有不同的含义。当讨论作业控制和信号时，我们谈及停止和继续执行作业。但是，终端驱动程序一直使用术语“停止”表示用Ctrl+S字符停止终端输出，为了继续启动该终端输出，则用Ctrl+Q字符。为此，终端驱动程序称产生交互式停止信号的字符为挂起字符，而非停止字符。

SIGTTIN 当一个后台进程组中的进程试图读其控制终端时，终端驱动程序产生此信号（见9.8节中对此问题的讨论）。在下列特殊情形下不产生此信号：(a) 读进程忽略或阻塞此信号，(b) 读进程所属的进程组是孤儿进程组，此时读操作返回出错，并将errno设置为EIO。

296

SIGTTOU 当一个后台进程组中的进程试图写到其控制终端时产生此信号（见9.8节对此主题的讨论）。与上面所述的SIGTTIN信号不同，一个进程可以选择允许后台进程写到控制终端。第18章将讨论如何更改此选项。

如果不允许后台进程写，则与SIGTTIN相似，也有两种特殊情况：(a) 写进程忽略或阻塞此信号，(b) 写进程所属进程组是孤儿进程组。在这两种情况下不产生此信号，写操作返回出错，并将errno设置为EIO。

不论是否允许后台进程写，某些除写以外的下列终端操作也能产生此信号：tcsetattr、tcsendbreak、tcdrain、tcflush、tcflow以及tcsetpgrp。第18章将说明这些终端操作。

SIGURG 此信号通知进程已经发生一个紧急情况。在网络连接上接收到带外的数据时，可选择产生此信号。

SIGUSR1 这是一个用户定义的信号，可用于应用程序。

SIGUSR2 这是另一个用户定义的信号，与SIGUSR1相似，可用于应用程序。

SIGVTALRM 当一个由setitimer(2)函数设置的虚拟间隔时间到期时产生此信号。

SIGWAITING 此信号由Solaris线程库内部使用，不作它用。

SIGWINCH 内核维持与每个终端或伪终端相关联的窗口大小。进程可以用ioctl函数（见18.12节的说明）得到或设置窗口的大小。如果进程用ioctl的设置窗口大小命令更改了窗口大小，则内核产生SIGWINCH信号并将其送至前台进程组。

SIGXCPU Single UNIX Specification的XSI扩展支持资源限制的概念（见7.11节）。如果进程超过了其软CPU时间限制，则产生SIGXCPU信号。

297

在表10-1中，针对SIGXCPU的默认动作被标记为“终止并创建core文件”或“忽略”。不幸的是，该默认动作依赖于操作系统。Linux 2.4.22和Solaris 9支持的默认动作是“终止并创建core文件”；FreeBSD 5.2.1和Mac OS X 10.3支持的默认动作是“忽略”。Single UNIX Specification要求该默认动作是异常终止该进程，是否创建core文件则留给实现决定。

SIGXFSZ　如果进程超过了其软文件长度限制（见7.11节），则产生此信号。

如同SIGXCPU一样，针对SIGXFSZ的默认动作依赖于操作系统。Linux 2.4.22和Solaris 9对此信号的默认动作是“终止进程并创建core文件”。FreeBSD 5.2.1和Mac OS X 10.3支持的默认动作是“忽略”。Single UNIX Specification要求该默认动作是异常终止该进程，是否创建core文件则留给实现决定。

SIGXRES　此信号仅由Solaris定义。可选择使用此信号以通知进程超过了预配置的资源值。Solaris资源限制机制是一种通用设施，用于控制在独立应用程序集之间使用共享资源。

10.3 signal函数

UNIX系统的信号机制最简单的接口是signal函数。

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

返回值：若成功则返回信号以前的处理配置（见下），若出错则返回SIG_ERR

signal函数由ISO C定义。因为ISO C不涉及多进程、进程组以及终端I/O等，所以它对信号的定义非常含糊，以至于对UNIX系统而言几乎毫无用处。

从UNIX 系统V派生的实现支持signal函数，但该函数提供旧的不可靠信号语义（10.4节将说明这些旧语义）。提供此函数主要是为了向后兼容那些需要此旧语义的应用程序，新应用程序不应使用这些不可靠信号。

4.4BSD也提供signal函数，但它是按照sigaction函数定义的（10.14节将说明sigaction函数），所以在4.4BSD之下使用它提供新的可靠信号语义。FreeBSD和Mac OS X遵循此种策略。

Solaris 9植根于系统V和BSD，但在signal函数方面，它依从系统V语义。

Linux 2.4.22的signal语义依从BSD或者系统V，这取决于C函数库的版本，以及编译应用程序的方法。

因为signal的语义与实现有关，所以最好使用sigaction函数代替signal函数。在10.4节讨论sigaction函数时，提供了使用该函数的一个signal实现。本书中的所有实例均使用程序清单10-12中给出的signal函数。

298

*signo*参数是表10-1中的信号名。*func*的值是常量SIG_IGN、常量SIG_DFL或当接到此信号后要调用的函数的地址。如果指定SIG_IGN，则向内核表示忽略此信号（记住有两个信号SIGKILL和SIGSTOP不能忽略）。如果指定SIG_DFL，则表示接到此信号后的动作是系统默认动作（见表10-1中的最后1列）。当指定函数地址时，则在信号发生时，调用该函数，我们称这种处理为“捕捉”该信号。称此函数为信号处理程序（signal handler）或信号捕捉函数（signal-catching function）。

signal函数原型说明此函数需要两个参数，返回一个函数指针，而该指针所指向的函数

无返回值（void）。第一个参数*signo*是一个整数，第二个参数是函数指针，它所指向的函数需要一个整型参数，无返回值。signal的返回值是一个函数地址，该函数有一个整型参数（即最后的（int））。用自然语言来描述也就是要向信号处理程序传送一个整型参数，而它却无返回值。当调用signal设置信号处理程序时，第二个参数是指向该函数（也就是信号处理程序）的指针。signal的返回值则是指向之前的信号处理程序的指针。

很多系统用附加的依赖于实现的参数来调用信号处理程序。10.14节将对此作进一步说明。

本节开头所示的signal函数原型太复杂了，如果使用下面的typedef[Plauger 1992]，则可使其简单一些。

```
typedef void Sigfunc(int);
```

然后，可将signal函数原型写成

```
Sigfunc *signal(int, Sigfunc *);
```

我们已将此typedef包括在apue.h文件中（见附录B），并随本章中的函数一起使用。

如果查看系统的头文件<signal.h>，则很可能会找到下列形式的声明：

```
#define SIG_ERR   (void (*)())-1
#define SIG_DFL   (void (*)())0
#define SIG_IGN   (void (*)())1
```

这些常量可用于代替“指向函数的指针，该函数需要一个整型参数，而且无返回值”。signal的第二个参数及其返回值就可用它们表示。这些常量所使用的三个值不一定是-1，0和1。它们必须是三个值而决不能是任一可声明函数的地址。大多数UNIX系统使用上面所示的值。

实 例

程序清单10-1显示了一个简单的信号处理程序，它捕捉两个用户定义的信号并打印信号编号。10.10节将说明pause函数，它使调用进程在接到一个信号前挂起。 299

程序清单10-1 捕捉SIGUSR1和SIGUSR2的简单程序

```
#include "apue.h"

static void sig_usr(int);   /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
```

```
        err_dump("received signal %d\n", signo);
}
```

我们在后台调用该程序，并且用kill(1)命令将信号传送给它。注意，在UNIX中，杀死（kill）这个术语是不恰当的。kill(1)命令和kill(2)函数只是将一个信号送给一个进程或进程组。信号是否终止进程则取决于信号的类型，以及进程是否安排了捕捉该信号。

```
$ ./a.out &                       在后台启动进程
[1]      7216                     作业控制shell打印作业号和进程
$ kill -USR1 7216                 向该进程发送SIGUSR1
received SIGUSR1
$ kill -USR2 7216                 向该进程发送SIGUSR2
received SIGUSR2
$ kill 7216                       向该进程发送SIGTERM
[1]+  Terminated          ./a.out
```

因为执行程序清单10-1的进程不捕捉SIGTERM信号，而针对该信号的系统默认动作是终止，所以当向该进程发送SIGTERM信号后，该进程就会终止。□

1. 程序启动

当执行一个程序时，所有信号的状态都是系统默认或忽略。通常所有信号都被设置为它们的默认动作，除非调用exec的进程忽略该信号。确切地讲，exec函数将原先设置为要捕捉的信号都更改为它们的默认动作，其他信号的状态则不变（对于一个进程原先要捕捉的信号，当其执行一个新程序后，就自然不能再捕捉它了，因为信号捕捉函数的地址很可能在所执行的新程序文件中已无意义）。

300

一个具体例子是一个交互式shell如何处理针对后台进程的中断和退出信号。对于一个非作业控制shell，当在后台执行一个进程时，例如：

```
cc main.c &
```

shell自动将后台进程对中断和退出信号的处理方式设置为忽略。于是，当按中断键时就不会影响到后台进程。如果没有执行这样的处理，那么当按中断键时，它不但会终止前台进程，还会终止所有后台进程。

很多捕捉这两个信号的交互式程序具有下列形式的代码：

```
void sig_int(int), sig_quit(int);

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

这样处理后，仅当信号当前未被忽略时，进程才会捕捉它们。

从signal的这两个调用中也可以看到这种函数的限制：不改变信号的处理方式就不能确定信号的当前处理方式。我们将在本章的稍后部分说明使用sigaction函数可以确定一个信号的处理方式，而无需改变它。

2. 进程创建

当一个进程调用fork时，其子进程继承父进程的信号处理方式。因为子进程在开始时复制了父进程的存储映像，所以信号捕捉函数的地址在子进程中是有意义的。

10.4 不可靠的信号

在早期的UNIX版本（例如V7）中，信号是不可靠的。不可靠在这里指的是，信号可能会

丢失：一个信号发生了，但进程却可能一直不知道这一点。同时，进程对信号的控制能力也很差，它能捕捉信号或忽略它。有时用户希望通知内核阻塞一个信号：不要忽略该信号，在其发生时记住它，然后在进程做好准备时再通知它。这种阻塞信号的能力当时并不具备。

4.2BSD对信号机制进行了更改，提供了被称为可靠信号的机制。然后，SVR3也修改了信号机制，提供了另一套系统V可靠信号机制。POSIX.1选择了BSD模型作为其标准化的基础。

301

早期版本中的一个问题是在进程每次接到信号对其进行处理时，随即将该信号动作复位为默认值（在前面运行程序清单10-1时，我们只捕捉每种信号一次，从而回避了这一点）。在描述这些早期系统的编程书籍中，有一个经典实例，它与如何处理中断信号相关，其代码与下面所示的相似：

```
        int     sig_int();          /* my signal handling function */

        ...
        signal(SIGINT, sig_int); /* establish handler */
        ...

    sig_int()
    {
        signal(SIGINT, sig_int); /* reestablish handler for next time */
        ...                      /* process the signal ... */
    }
```

（由于早期的C语言版本不支持ISO C的void数据类型，所以将信号处理程序声明为int类型。）

这段代码的一个问题是：从信号发生之后到在信号处理程序中调用signal函数之前这段时间中有一个时间窗口。在此段时间中，可能发生另一次中断信号。第二个信号会导致执行默认动作，而针对中断信号的默认动作是终止该进程。这种类型的程序段在大多数情况下会正常工作，使得我们认为它们是正确无误的，而实际上并非如此。

这些早期系统的另一个问题是：在进程不希望某种信号发生时，它不能关闭该信号。进程能做的一切就是忽略该信号。有时希望通知系统“阻止下列信号发生，如果它们确实产生了，请记住它们。”能够显现这种缺陷的一个经典实例是下列程序段，它捕捉一个信号，然后设置一个表示该信号已发生的标志：

```
    int   sig_int_flag;            /* set nonzero when signal occurs */

    main()
    {
        int     sig_int();          /* my signal handling function */
        ...
        signal(SIGINT, sig_int); /* establish handler */
        ...
        while (sig_int_flag == 0)
            pause();               /* go to sleep, waiting for signal */
        ...
    }

    sig_int()
    {
        signal(SIGINT, sig_int); /* reestablish handler for next time */
        sig_int_flag = 1;          /* set flag for main loop to examine */
    }
```

302

其中，进程调用pause函数使自己休眠，直至捕捉到一个信号。当捕捉到信号时，信号处理程

序将标志sig_int_flag设置为非0值。从信号处理程序返回后，内核自动将该进程唤醒，它检测到该标志为非0，然后执行它所需做的工作。但是这里也有一个时间窗口，在此窗口中操作可能失误。如果在测试sig_int_flag之后和调用pause之前发生信号，则此进程在调用pause时入睡，并且长眠不醒（假定此信号不会再次产生）。于是，这次发生的信号也就丢失了。这是另一个例子，某段代码并不正确，但是大多数时间却能正常工作。要查找并排除这种类型的问题很困难。

10.5　中断的系统调用

早期UNIX系统的一个特性是：如果进程在执行一个低速系统调用而阻塞期间捕捉到一个信号，则该系统调用就被中断不再继续执行。该系统调用返回出错，其errno被设置为EINTR。这样处理的理由是：因为一个信号发生了，进程捕捉到了它，这意味着已经发生了某种事情，所以是个应当唤醒阻塞的系统调用的好机会。

在这里，我们必须区分系统调用和函数。当捕捉到某个信号时，被中断的是内核中执行的系统调用。

为了支持这种特性，将系统调用分成两类：低速系统调用和其他系统调用。低速系统调用是可能会使进程永远阻塞的一类系统调用，它们包括：

- 在读某些类型的文件（管道、终端设备以及网络设备）时，如果数据并不存在则可能会使调用者永远阻塞。
- 在写这些类型的文件时，如果不能立即接受这些数据，则也可能会使调用者永远阻塞。
- 打开某些类型文件，在某种条件发生之前也可能会使调用者阻塞（例如，打开终端设备，它要等待直到所连接的调制解调器应答了电话）。
- pause（按照定义，它使调用进程休眠直至捕捉到一个信号）和wait函数。
- 某些ioctl操作。
- 某些进程间通信函数（见第15章）。

在这些低速系统调用中，一个值得注意的例外是与磁盘I/O有关的系统调用。虽然读、写一个磁盘文件可能暂时阻塞调用者（在磁盘驱动器将请求排入队列，然后在适当时间执行请求期间），但是除非发生硬件错误，I/O操作总会很快返回，并使调用者不再处于阻塞状态。

可以用中断系统调用这种方法来处理的一个例子是：一个进程启动了读终端操作，而使用该终端设备的用户却离开该终端很长时间。在这种情况下进程可能处于阻塞状态几个小时甚至数天，除非系统停机，否则一直如此。

303

对于中断的read、write系统调用，POSIX.1的语义在该标准的2001版有所改变。对于如何处理已read、write部分数据量的相应系统调用，早期版本允许实现进行选择。如若read系统调用已接收并传送数据至应用程序缓冲区，但尚未接收到应用程序请求的全部数据，此时被中断，操作系统可以认为该系统调用失败，并将errno设置为EINTR；另一种处理方式是允许该系统调用成功返回，返回已接收到的部分数据量。与此类似，如若write已传输了应用程序缓冲区中的部分数据，然后被中断，操作系统可以认为该系统调用失败，并将errno设置为EINTR；另一种处理方式是允许该系统调用成功返回，返回已写的部分数据量。历史上，从系统V派生的实现，将这种系统调用视为失败，而BSD派生的实现则处理为部分成功返回。POSIX.1标准的2001版采用BSD风格的语义。

与被中断的系统调用相关的问题是必须显式地处理出错返回。典型的代码序列（假定进行

一个读操作，它被中断，我们希望重新启动它）可能如下所示：

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again;     /* just an interrupted system call */
        /* handle other errors */
    }
```

为了帮助应用程序使其不必处理被中断的系统调用，4.2BSD引入了某些被中断系统调用的自动重启动。自动重启动的系统调用包括ioctl、read、readv、write、writev、wait和waitpid。正如前述，其中前5个函数只有对低速设备进行操作时才会被信号中断。而wait和waitpid在捕捉到信号时总是被中断。因为这种自动重启动的处理方式也会带来问题，所以某些应用程序并不希望这些函数被中断后重启动。为此4.3BSD允许进程基于每个信号禁用此功能。

POSIX.1允许实现重启动系统调用，但这并不是必需的。Single UNIX Specification将SA_RESTART定义为对sigaction的XSI扩展，以允许应用程序要求重启动被中断的系统调用。

系统V的默认工作方式是从不重启动系统调用。而BSD则重启动被信号中断的系统调用。FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3的默认方式是重启动由信号中断的系统调用。Solaris 9的默认方式是出错返回，并将errno设置为EINTR。

4.2BSD引入自动重启动功能的一个理由是：有时用户并不知道所使用的输入、输出设备是否是低速设备。如果我们编写的程序可以用交互方式运行，则它可能读、写低速终端设备。如果在程序中捕捉信号，而且系统并不提供重启动功能，则对每次读、写系统调用都要进行是否出错返回的测试，如果是被中断的，则再调用读、写系统调用。

表10-2列出了几种实现所提供的与信号有关的函数及其语义。

表10-2 几种信号实现所提供的功能

函数	系统	保持安装信号处理程序	阻塞信号的能力	被中断的系统调用自动重启动?
signal	ISO C、POSIX.1	未说明	未说明	未说明
	V7、SVR2、SVR3、SVR4、Solaris			决不
	4.2BSD	•	•	总是
	4.3BSD、4.4BSD、FreeBSD、Linux、Mac OS X	•	•	默认
sigset	XSI	•	•	未说明
	SVR3、SVR4、Linux、Solaris	•	•	决不
sigvec	4.2BSD	•	•	总是
	4.3BSD、4.4BSD、FreeBSD、Mac OS X	•	•	默认
sigaction	POSIX.1	•	•	未说明
	XSI、4.4BSD、SVR4、FreeBSD、Mac OS X、Linux、Solaris	•	•	可选

我们没有讨论旧的sigset和sigvec函数。它们已由sigaction函数替代；仅仅为了完整性我们才将这两个函数包含在表10-2中。与之对照，某些实现将signal函数提升为sigaction的简化接口。

304

应当了解，其他厂商提供的UNIX系统可能不同于表10-2中所示的情况。例如，SunOS 4.1.2中的sigaction的默认方式是重启动被中断的系统调用，这与表10-2中所列的各平台不同。

在程序清单10-12中，提供了我们自己的signal函数版本，它自动地试图重启动被中断的系统调用（除SIGALRM信号外）。在程序清单10-13中则提供了另一个函数signal_intr，它从不尝试进行重启动。

14.5节说明select和poll函数时还将涉及被中断的系统调用的更多知识。

10.6 可重入函数

进程捕捉到信号并对其进行处理时，进程正在执行的指令序列就被信号处理程序临时中断，它首先执行该信号处理程序中的指令。如果从信号处理程序返回（例如没有调用exit或longjmp），则继续执行在捕捉到信号时进程正在执行的正常指令序列（这类似于发生硬件中断时所做的）。但在信号处理程序中，不能判断捕捉到信号时进程在何处执行。如果进程正在执行malloc，在其堆中分配另外的存储空间，而此时由于捕捉到信号而插入执行该信号处理程序，其中又调用malloc，这时会发生什么？又例如若进程正在执行getpwnam（见6.2节）这种将其结果存放在静态存储单元中的函数，其间插入执行信号处理程序，它又调用这样的函数，这时又会发生什么呢？在malloc例子中，可能会对进程造成破坏，因为malloc通常为它所分配的存储区维护一个链接表，而插入执行信号处理程序时，进程可能正在更改此链接表。在getpwnam的例子中，返回给正常调用者的信息可能被返回给信号处理程序的信息覆盖。

305

Single UNIX Specification说明了保证可重入的函数。表10-3列出了这些可重入函数。

表10-3　信号处理程序可以调用的可重入函数

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf
aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit&_exit	listen	sendmsg	socketpair	write

没有列入表10-3中的大多数函数是不可重入的，其原因为：(a) 已知它们使用静态数据结构，(b) 它们调用malloc或free，或(c) 它们是标准I/O函数。标准I/O库的很多实现都以不可重入方式使用全局数据结构。注意，即使在本书的某些实例中，信号处理程序也调用了printf函数，但这并不保证产生所期望的结果，信号处理程序可能中断主程序中的printf函数调用。

应当了解即使信号处理程序调用的是列于表10-3中的函数，但是由于每个线程只有一个errno变量（回忆1.7节对errno和线程的讨论），所以信号处理程序可能会修改其原先值。考虑一个信号处理程序，它恰好在main刚刚设置errno之后被调用。例如，如果该信号处理程序调用read这类函数，则它可能更改errno的值，从而取代了刚刚由main设置的值。因此，作为一个通用的规则，当在信号处理程序中调用表10-3中列出的函数时，应当在其前保存，在其后恢复errno。（应当了解，经常被捕捉到的信号是SIGCHLD，其信号处理程序通常要调用一种wait函数，而各种wait函数都能改变errno。）

306

注意，表10-3没有包括longjmp（7.10节）和siglongjmp（10.15节）。这是因为主例程以非可重入方式正在更新数据结构时可能产生信号。如果不是从信号处理程序返回而是调用siglongjmp，那么该数据结构可能是部分更新的。如果应用程序将要做更新全局数据结构这样的事情，同时要捕捉某些信号，而这些信号的处理程序又会引起执行sigsetjmp，则在更新这种数据结构时要阻塞此类信号。

实例

在程序清单10-2中，信号处理程序my_alarm调用不可重入函数getpwnam，而my_alarm每秒钟被调用一次。10.10节中将说明alarm函数。在该程序中调用alarm函数使得每秒产生一次SIGALRM信号。

程序清单10-2 在信号处理程序中调用不可重入函数

```
#include "apue.h"
#include <pwd.h>

static void
my_alarm(int signo)
{
    struct passwd   *rootptr;

    printf("in signal handler\n");
    if ((rootptr = getpwnam("root")) == NULL)
            err_sys("getpwnam(root) error");
    alarm(1);
}

int
main(void)
{
    struct passwd   *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);
    for ( ; ; ) {
        if ((ptr = getpwnam("sar")) == NULL)
            err_sys("getpwnam error");
        if (strcmp(ptr->pw_name, "sar") != 0)
            printf("return value corrupted!, pw_name = %s\n",
```

```
                        ptr->pw_name);
        }
    }
```

运行该程序时，其结果具有随意性。通常，在信号处理程序经多次迭代返回时，该程序将由SIGSEGV信号终止。检查core文件，从中可以看到main函数已调用getpwnam，而且当信号处理程序调用此同一函数时，某些内部指针出了问题。偶然，此程序会运行若干秒，然后因产生SIGSEGV信号而终止。在捕捉到信号后，若main函数仍正确运行，其返回值却有时错误，有时正确。有一次在Mac OS X上运行该程序时曾经打印出来自malloc库例程的警告信息，声称正释放的指针是未经malloc分配的。

307

从此实例中可以看出，若在信号处理程序中调用一个不可重入函数，则其结果是不可预见的。 □

10.7　SIGCLD语义

SIGCLD和SIGCHLD这两个信号很容易被混淆。SIGCLD（没有H）是系统V的一个信号名，其语义与名为SIGCHLD的BSD信号不同。POSIX.1则采用BSD的SIGCHLD信号。

BSD的SIGCHLD信号语义与其他信号的语义相类似。子进程状态改变后产生此信号，父进程需要调用一个wait函数以确定发生了什么。

由于历史原因，系统V处理SIGCLD信号的方式不同于其他信号。如果用signal或sigset（早期设置信号配置的与SRV3兼容的函数）设置信号配置，则基于SVR4的系统继续了这一具有问题色彩的传统（即兼容性限制）。对于SIGCLD的早期处理方式如下：

(1) 如果进程特地设置该信号的配置为SIG_IGN，则调用进程的子进程将不产生僵死进程。注意，这与其默认动作（SIG_DFL）“忽略”（见图10-1）不同。代之以在子进程终止时，将其状态丢弃。如果调用进程随后调用一个wait函数，那么它将阻塞到所有子进程都终止，然后该wait会返回-1，并将其errno设置为ECHILD（此信号的默认配置是忽略，但这不会造成上述语义起作用。代之以我们必须特地指定其配置为SIG_IGN)。

POSIX.1并未说明在SIGCHLD被忽略时应产生的后果，所以这种运行行为是允许的。Single UNIX Specification包括了一个XSI扩展，它规定对于SIGCHLD支持这种运行行为。

如果SIGCHLD被忽略，4.4BSD总是产生僵死子进程。如果要避免僵死子进程，则必须等待子进程。FreeBSD 5.2.1对此的处理方式与4.4BSD相同。但是，Mac OS X 10.3在SIGCHLD被忽略时，并不创建僵死子进程。

在SVR4中，如果调用signal或sigset将SIGCHLD的配置设置为忽略，则决不会产生僵死子进程。Linux 2.4.22和Solaris 9在此方面追随SVR4。

使用sigaction可设置SA_NOCLDWAIT标志（见表10-5）以避免子进程僵死。本书讨论的四种平台都支持这一动作。

308

(2) 如果将SIGCLD的配置设置为捕捉，则内核立即检查是否有子进程准备好被等待，如果是这样，则调用SIGCLD处理程序。

第(2)项改变了为此信号编写处理程序的方法。这一点可在下面的实例中看到。

实　例

10.4节曾提到进入信号处理程序后，首先要调用signal函数以重新设置此信号处理程序

(在信号被复位回其默认值时，它可能被丢失，立即重新设置可以减少此窗口时间)。程序清单10-3显示了这一点。但此程序不能在某些平台上正常工作。如果在传统的系统V平台（例如OpenServer 5或UnixWare 7）上编译并运行此程序，则其输出是一行行地不断重复“SIGCLD received”。最后进程用完其栈空间并异常终止。

程序清单10-3 不能正常工作的系统V SIGCLD处理程序

```
#include    "apue.h"
#include    <sys/wait.h>

static void sig_cld(int);

int
main()
{
    pid_t   pid;

    if (signal(SIGCLD, sig_cld) == SIG_ERR)
        perror("signal error");
    if ((pid = fork()) < 0) {
        perror("fork error");
    } else if (pid == 0) {      /* child */
        sleep(2);
        _exit(0);
    }
    pause();    /* parent */
    exit(0);
}

static void
sig_cld(int signo)  /* interrupts pause() */
{
    pid_t   pid;
    int     status;

    printf("SIGCLD received\n");
    if (signal(SIGCLD, sig_cld) == SIG_ERR) /* reestablish handler */
        perror("signal error");
    if ((pid = wait(&status)) < 0)      /* fetch child status */
        perror("wait error");
    printf("pid = %d\n", pid);
}
```

309

因为基于BSD的系统通常并不支持早期系统V SIGCLD的语义，所以FreeBSD 5.2.1和Mac OS X 10.3并没有出现此问题。Linux 2.4.22也没有出现此问题，其原因是，虽然SIGCLD和SIGCHLD定义为同一值，但当一进程安排捕捉SIGCHLD，并且已经有进程准备好由其父进程等待时，该系统并不调用SIGCHLD信号的处理程序。另一方面，Solaris 9在此种情况确实调用该信号处理程序，但在内核中增加了避免此问题的代码。

虽然本书说明的所有四种平台都解决了这一问题，但是应当理解没有解决这一问题的平台（例如UnixWare）依然存在。

此程序的问题是：在信号处理程序的开始处调用signal，按照上述第2项，内核检查是否有需要等待的子进程（因为我们正在处理一个SIGCLD，所以确实有这种子进程），所以它产生另一个对信号处理程序的调用。信号处理程序调用signal，整个过程再次重复。

为了解决这一问题，应当在调用wait取到子进程的终止状态后再调用signal。此时仅当其他子进程终止时，内核才会再次产生此种信号。

如果为SIGCHLD建立了一个信号处理程序，又存在一个已终止但父进程尚未等待它的进程，则是否会产生信号？POSIX.1对此没有作说明。这样就允许前面所述的工作方式。但是，POSIX.1在信号发生时并没有将信号配置复位为其默认值（假定正调用POSIX.1的sigaction函数设置其配置），于是在SIGCHLD处理程序中也就不必再为该信号建立一个信号处理程序。□

务必了解你所用的系统中SIGCHLD信号的语义。也应了解在某些系统中#define SIGCHLD为SIGCLD或反之。更改这种信号的名字使你可以编译为另一个系统编写的程序，但是如果这一程序使用该信号的另一种语义，则这样的程序也不能工作。

在本书说明的四种平台上，SIGCLD等价于SIGCHLD。

10.8　可靠信号术语和语义

我们需要先定义一些在讨论信号时会用到的术语。首先，当引发信号的事件发生时，为进程产生一个信号（或向进程发送一个信号）。事件可以是硬件异常（例如，除以0）、软件条件（例如，alarm计时器超时）、终端产生的信号或调用kill函数。在产生了信号时，内核通常在进程表中设置一个某种形式的标志。

当对信号采取了这种动作时，我们说向进程递送了一个信号。在信号产生（generation）和递送（delivery）之间的时间间隔内，称信号是未决的（pending）。

310

进程可以选用信号递送阻塞。如果为进程产生了一个选择为阻塞的信号，而且对该信号的动作是系统默认动作或捕捉该信号，则为该进程将此信号保持为未决状态，直到该进程(a) 对此信号解除了阻塞，或者(b) 将对此信号的动作更改为忽略。内核在递送一个原来被阻塞的信号给进程时（而不是在产生该信号时），才决定对它的处理方式。于是进程在信号递送给它之前仍可改变对该信号的动作。进程调用sigpending函数（见10.13节）来判定哪些信号是设置为阻塞并处于未决状态的。

如果在进程解除对某个信号的阻塞之前，这种信号发生了多次，那么将如何呢？POSIX.1允许系统递送该信号一次或多次。如果递送该信号多次，则称对这些信号进行了排队。但是除非支持POSIX.1实时扩展，否则大多数UNIX并不对信号排队。代之以UNIX内核只递送这种信号一次。

SVR2的手册页称，在进程执行SIGCLD信号处理程序期间，该信号是用排队方式处理的，虽然在概念层次上这可能是真的，但实际并非如此。代之以，内核按10.7节中所述方式产生此信号。SVR3的手册页对此做了修改，它指明在进程执行SIGCLD信号处理程序期间，忽略SIGCLD信号。SVR4手册页删除了有关部分，也就是说，在进程执行SIGCLD信号处理程序期间，如若又产生了SIGCLD信号，SVR4手册页对此未作任何说明。

AT&T[1990e]中的SVR4 sigaction(2)手册页称SA_SIGINFO标志（见图10-16）使信号可靠地排队，这也不正确。表面上内核部分地实现了此功能，但在SVR4中并不起作用。令人不可思议的是，SVID对这种可靠排队列并未作同样的声明。

如果有多个信号要递送给一个进程，那么将如何呢？POSIX.1并没有规定这些信号的递送顺序。但是POSIX.1的Rationale建议：在其他信号之前递送与进程当前状态有关的信号，例如SIGSEGV。

每个进程都有一个信号屏蔽字（signal mask），它规定了当前要阻塞递送到该进程的信号集。对于每种可能的信号，该屏蔽字中都有一位与之对应。对于某种信号，若其对应位已设置，则

它当前是被阻塞的。进程可以调用sigprocmask（在10.12节中说明）来检测和更改其当前信号屏蔽字。

信号数量可能会超过整型所包含的二进制位数，因此POSIX.1定义了一个新数据类型sigset_t，用于保存一个信号集。例如，信号屏蔽字就存放在这些信号集的一个中。10.11节将说明对信号集进行操作的5个函数。

10.9 kill和raise函数

kill函数将信号发送给进程或进程组。raise函数则允许进程向自身发送信号。

> raise原来是由ISO C定义的。后来，为了与ISO C标准保持一致，POSIX.1也包括了该函数。但是POSIX.1扩展了raise的规范，使其可处理线程（12.8节中讨论线程如何与信号交互）。因为ISO C并不涉及多进程，所以它不能定义如kill这样要有一个进程ID作为其参数的函数。

311

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

两个函数返回值：若成功则返回0，若出错则返回-1

调用

```
raise(signo);
```

等价于调用

```
kill(getpid(), signo);
```

kill的*pid*参数有4种不同的情况：

- *pid* > 0　将该信号发送给进程ID为*pid*的进程。
- *pid* == 0　将该信号发送给与发送进程属于同一进程组的所有进程（这些进程的进程组ID等于发送进程的进程组ID），而且发送进程具有向这些进程发送信号的权限。注意，这里用的术语“所有进程”不包括实现定义的系统进程集。对于大多数UNIX系统，系统进程集包括内核进程以及init(pid 1)。
- *pid* < 0　将该信号发送给其进程组ID等于*pid*的绝对值，而且发送进程具有向其发送信号的权限。如上所述，“所有进程集”并不包括某些系统进程。
- *pid* == −1　将该信号发送给发送进程有权限向它们发送信号的系统上的所有进程。如上所述，“进程集”不包括某些系统进程。

上面曾提及，进程将信号发送给其他进程需要权限。超级用户可将信号发送给任一进程。对于非超级用户，其基本规则是发送者的实际或有效用户ID必须等于接收者的实际或有效用户ID。如果实现支持_POSIX_SAVED_IDS（如POSIX.1现在要求的那样），则检查接收者的保存的设置用户ID（而不是其有效用户ID）。在对权限进行测试时也有一个特例：如果被发送的信号是SIGCONT，则进程可将它发送给属于同一会话的任何其他进程。

POSIX.1将编号为0的信号定义为空信号。如果*signo*参数是0，则kill仍执行正常的错误检查，但不发送信号。这常被用来确定一个特定进程是否仍旧存在。如果向一个并不存在的进程发送空信号，则kill返回−1，并将errno设置为ESRCH。但是，应当了解，UNIX系统在经过一段时间后会重新使用进程ID，所以一个现有的具有所给定进程ID的进程并不一定就是你想要

的进程。

还应理解的是，对于进程是否存在的这种测试不是原子操作。在kill向调用者返回测试结果时，原来存在的被测试进程此时可能已经终止，所以这种测试并无多大价值。

312

如果调用kill为调用进程产生信号，而且此信号是不被阻塞的，那么在kill返回之前，就会将*signo*或者某个其他未决的非阻塞信号传送至该进程。（对于线程而言，还有一些附加条件；详细情况见12.8节。）

10.10 alarm和pause函数

使用alarm函数可以设置一个计时器，在将来某个指定的时间该计时器会超时。当计时器超时时，产生SIGALRM信号。如果不忽略或不捕捉此信号，则其默认动作是终止调用该alarm函数的进程。

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

                                返回值：0或以前设置的闹钟时间的余留秒数
```

其中，参数*seconds*的值是秒数，经过了指定的*seconds*秒后会产生信号SIGALRM。要了解的是，经过了指定的秒数后，信号由内核产生，由于进程调度的延迟，所以进程得到控制从而能够处理该信号还需一些时间。

早期的UNIX系统实现曾提出警告，这种信号可能比预定值提前1秒发送。POSIX.1则不允许这样做。

每个进程只能有一个闹钟时钟。如果在调用alarm时，以前已为该进程设置过闹钟时钟，而且它还没有超时，则将该闹钟时钟的余留值作为本次alarm函数调用的值返回。以前登记的闹钟时钟则被新值代替。

如果有以前为进程登记的尚未超过的闹钟时钟，而且本次调用的*seconds*值是0，则取消以前的闹钟时钟，其余留值仍作为alarm函数的返回值。

虽然SIGALRM的默认动作是终止进程，但是大多数使用闹钟的进程会捕捉此信号。如果此时进程要终止，则在终止之前它可以执行所需的清理操作。如果我们想捕捉SIGALRM信号，则必须在调用alarm之前设置该信号的处理程序。如果我们先调用alarm，然后在我们能够设置SIGALRM处理程序之前已接收到该信号，那么进程将终止。

pause函数使调用进程挂起直至捕捉到一个信号。

```
#include <unistd.h>

int pause(void);

                                返回值：-1，并将errno设置为EINTR
```

只有执行了一个信号处理程序并从其返回时，pause才返回。在这种情况下，pause返回-1，并将errno设置为EINTR。

313

实例

使用alarm和pause，进程可使自己休眠一段指定的时间。程序清单10-4中的sleep1函

数提供这种功能（但是它有一些问题，我们很快就会看到）。

程序清单10-4 sleep的简单而不完整的实现

```
#include    <signal.h>
#include    <unistd.h>

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);           /* start the timer */
    pause();                /* next caught signal wakes us up */
    return(alarm(0));       /* turn off timer, return unslept time */
}
```

程序中的sleep1函数看起来与将在10.19节中说明的sleep函数类似，但这种简单实现有下列三个问题：

(1) 如果在调用sleep1之前，调用者已设置了闹钟，则它会被sleep1函数中的第一次alarm调用擦除。可用下列方法更正这一点：检查第一次调用alarm的返回值，如其小于本次调用alarm的参数值，则只应等到上次设置的闹钟超时。如果上次设置闹钟的超时时间晚于本次设置值，则在sleep1函数返回之前，复位此闹钟，使其在上次闹钟的设定时间再次发生超时。

(2) 该程序中修改了对SIGALRM的配置。如果编写了一个函数供其他函数调用，则在该函数被调用时先要保存原配置，在该函数返回前再恢复原配置。更正这一点的方法是：保存signal函数的返回值，在返回前复位原配置。

(3) 在第一次调用alarm和调用pause之间有一个竞争条件。在一个繁忙的系统中，可能alarm在调用pause之前超时，并调用了信号处理程序。如果发生这种情况，则在调用pause后，如果没有捕捉到其他信号，则调用者将永远被挂起。

sleep的早期实现与程序清单10-4类似，但更正了问题(1)和(2)。有两种方法可以更正问题(3)。第一种方法是使用setjmp，下一个实例将说明这种方法。另一种方法是使用sigprocmask和sigsuspend，10.19节将说明这种方法。 □ 314

SVR2中的sleep实现使用了setjmp和longjmp（见7.10节），以避免前一个实例问题(3)中所说明的竞争条件。此函数的一个简单版本称为sleep2，示于程序清单10-5中（为了缩短实例长度，程序中没有处理上面所说的问题(1)和(2)）。

程序清单10-5 sleep的另一个（不完善）实现

```
#include    <setjmp.h>
#include    <signal.h>
#include    <unistd.h>

static jmp_buf  env_alrm;
```

```
static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alrm) == 0) {
        alarm(nsecs);       /* start the timer */
        pause();            /* next caught signal wakes us up */
    }
    return(alarm(0));       /* turn off timer, return unslept time */
}
```

在此函数中，程序清单10-4具有的竞争条件已被避免。即使pause从未执行，在发生SIGALRM时，sleep2函数也会返回。

但是，sleep2函数中却有另一个难以察觉的问题，它涉及与其他信号的交互。如果SIGALRM中断了某个其他信号处理程序，则调用longjmp会提早终止该信号处理程序。程序清单10-6显示了这种情况。SIGINT处理程序中包含了for循环语句，它在作者所用系统上的执行时间超过5秒钟，也就是大于sleep2的参数值，这正是我们想要的。将整型变量k声明为volatile，这样就阻止了优化编译器丢弃循环语句。执行程序清单10-6得到：

```
$ ./a.out
^?                          我们键入中断字符
sig_int starting
sleep2 returned: 0
```

从中可见sleep2函数所引起的longjmp使另一个信号处理程序sig_int提早终止，即使它未完成也会如此。如果将SVR2的sleep函数与其他信号处理程序一起使用，就可能碰到这种情况。见习题10.3。 □

315

程序清单10-6　在一个捕捉其他信号的程序中调用sleep2

```
#include "apue.h"

unsigned int    sleep2(unsigned int);
static void     sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);
    exit(0);
}

static void
sig_int(int signo)
{
    int             i, j;
```

```
    volatile int    k;

    /*
     * Tune these loops to run for more than 5 seconds
     * on whatever system this test program is run.
     */
    printf("\nsig_int starting\n");
    for (i = 0; i < 300000; i++)
        for (j = 0; j < 4000; j++)
            k += i * j;
    printf("sig_int finished\n");
}
```

有关sleep1和sleep2函数的这两个实例的目的是告诉我们在涉及信号时需要有精细而周到的考虑。下面几节将说明解决这些问题的方法，使我们能够可靠地、在不影响其他代码段的情况下处理信号。



除了用来实现sleep函数外，alarm还常用于对可能阻塞的操作设置时间上限值。例如，程序中有一个读低速设备的可能阻塞的操作（见10.5节），我们希望超过一定时间量后就停止执行该操作。程序清单10-7实现了这一点，它从标准输入读一行，然后将其写到标准输出上。 316

程序清单10-7 具有超时限制的read调用

```
#include "apue.h"

static void sig_alrm(int);

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to interrupt the read */
}
```

这种代码序列在很多UNIX应用程序中都能见到，但是这种程序有两个问题：

(1) 程序清单10-7具有与程序清单10-4相同的问题：在第一次alarm调用和read调用之间有一个竞争条件。如果内核在这两个函数调用之间使进程阻塞，而其时间长度又超过闹钟时间，则read可能永远阻塞。大多数这种类型的操作使用较长的闹钟时间，例如1分钟或更长一点，

使这种问题不会发生，但无论如何这是一个竞争条件。

(2) 如果系统调用是自动重启动的，则当从SIGALRM信号处理程序返回时，read并不被中断。在这种情形下，设置时间限制不起作用。

在这里我们确实需要中断低速系统调用。POSIX.1并未提供一种可移植的方法来实现这一点，但是，Single UNIX Specification的XSI扩展却做到了这一点。我们将在10.14节对此进行详细讨论。 □

让我们用longjmp重新实现前面的实例（见程序清单10-8）。使用这种方法则无需担心一个慢速的系统调用是否被中断。

317

程序清单10-8　使用longjmp，带超时限制调用read

```
#include "apue.h"
#include <setjmp.h>

static void     sig_alrm(int);
static jmp_buf  env_alrm;

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    if (setjmp(env_alrm) != 0)
        err_quit("read timeout");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

不管系统是否重新启动中断的系统调用，该程序都会如所预期的那样工作。但是要知道，该程序仍旧有和程序清单10-5中相同的与其他信号处理程序交互的问题。 □

如果要对I/O操作设置时间限制，则如上所示可以使用longjmp，当然也要清楚它可能有与其他信号处理程序交互的问题。另一种选择是使用select或poll函数，14.5.1节和14.5.2节将对它们进行说明。

10.11　信号集

我们需要有一个能表示多个信号——信号集（signal set）的数据类型。我们将在诸如

sigprocmask（下一节中说明）之类的函数中使用这种数据类型，以便告诉内核不允许发生该信号集中的信号。如前所述，信号种类数目可能超过一个整型量所包含的位数，所以一般而言，不能用整型量中的一位代表一种信号，也就是不能用一个整型量表示信号集。POSIX.1定义了数据类型sigset_t以包含一个信号集，并且定义了下列五个处理信号集的函数。 318

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);

                                    四个函数的返回值：若成功则返回0，若出错则返回-1

int sigismember(const sigset_t *set, int signo);

                                    返回值：若真则返回1，若假则返回0，若出错则返回-1
```

函数sigemptyset初始化由*set*指向的信号集，清除其中所有信号。函数sigfillset初始化由set指向的信号集，使其包括所有信号。所有应用程序在使用信号集前，要对该信号集调用sigemptyset或sigfillset一次。这是因为C编译器将把未赋初值的外部和静态变量都初始化为0，而这是否与给定系统上信号集的实现相对应却并不清楚。

一旦已经初始化了一个信号集，以后就可在该信号集中增、删特定的信号。函数sigaddset将一个信号添加到现有集中，sigdelset则从信号集中删除一个信号。对所有以信号集作为参数的函数，我们总是以信号集地址作为向其传送的参数。

实例

如果实现的信号数目少于一个整型量所包含的位数，则可用一位代表一个信号的方法实现信号集。例如，在本书的后续部分，我们都假定一种实现有31种信号和32位整型量。sigemptyset函数将整型量设置为0，sigfillset函数则将整型量中的各个位都设置为1。这两个函数可以在<signal.h>头文件中实现为宏：

```
#define sigemptyset(ptr)   (*(ptr) = 0)
#define sigfillset(ptr)    (*(ptr) = ~(sigset_t)0, 0)
```

注意，除了设置信号集中各位为1外，sigfillset必须返回0，所以使用C语言的逗号运算符，它将逗号运算符后的值作为表达式的值返回。

使用这种实现，sigaddset打开一位（将该位设置为1），sigdelset则关闭一位（将该位设置为0），sigismember测试一指定位。因为没有编号为0的信号，所以从信号编号中减去1以得到要处理位的位编号数。程序清单10-9实现了这些函数。 319

程序清单10-9 sigaddset、sigdelset和sigismember的实现

```
#include    <signal.h>
#include    <errno.h>

/* <signal.h> usually defines NSIG to include signal number 0 */
#define SIGBAD(signo)   ((signo) <= 0 || (signo) >= NSIG)

int
```

```
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);       /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &= ~(1 << (signo - 1));    /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    return((*set & (1 << (signo - 1))) != 0);
}
```

也可将这三个函数在`<signal.h>`中实现为单行宏，但是POSIX.1要求检查信号编号参数的有效性，如果无效则设置errno。在宏中实现这一点比在函数中要困难。

10.12 sigprocmask函数

10.8节曾提及一个进程的信号屏蔽字规定了当前阻塞而不能递送给该进程的信号集。调用函数sigprocmask可以检测或更改其信号屏蔽字，或者在一个步骤中同时执行这两个操作。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

返回值：若成功则返回0，若出错则返回-1

320

首先，若*oset*是非空指针，那么进程的当前信号屏蔽字通过*oset*返回。

其次，若*set*是一个非空指针，则参数*how*指示如何修改当前信号屏蔽字。表10-4说明了*how*可选用的值。SIG_BLOCK是“或”操作，而SIG_SETMASK则是赋值操作。注意，不能阻塞SIGKILL和SIGSTOP信号。

表10-4 用sigprocmask更改当前信号屏蔽字的方法

how	说　明
SIG_BLOCK	该进程新的信号屏蔽字是其当前信号屏蔽字和*set*指向信号集的并集。*set*包含了我们希望阻塞的附加信号
SIG_UNBLOCK	该进程新的信号屏蔽字是其当前信号屏蔽字和*set*所指向信号集补集的交集。*set*包含了我们希望解除阻塞的信号
SIG_SETMASK	该进程新的信号屏蔽字将被*set*指向的信号集的值代替

如果*set*是空指针，则不改变该进程的信号屏蔽字，*how*的值也无意义。

在调用sigprocmask后如果有任何未决的、不再阻塞的信号，则在sigprocmask返回前，至少会将其中一个信号递送给该进程。

sigprocmask是仅为单线程的进程定义的。为处理多线程的进程中信号的屏蔽，提供了另一个单独的函数。我们将在12.8节中对此进行讨论。

实 例

程序清单10-10显示了一个函数，它打印调用进程的信号屏蔽字中信号的名称。程序清单10-14和程序清单10-15将调用此函数。

程序清单10-10 为进程打印信号屏蔽字

```
#include "apue.h"
#include <errno.h>

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);

    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");

    /* remaining signals can go here  */

    printf("\n");
    errno = errno_save;
}
```

321

为了节省空间，没有对表10-1中列出的每一种信号测试该屏蔽字（见习题10.9）。

10.13 sigpending函数

sigpending函数返回信号集，其中的各个信号对于调用进程是阻塞的而不能递送，因而也一定是当前未决的。该信号集通过*set*参数返回。

```
#include <signal.h>

int sigpending(sigset_t *set);
```

返回值：若成功则返回0，若出错则返回−1

实 例

程序清单10-11使用了很多前面说明过的信号功能。

程序清单10-11　信号设置和sigprocmask实例

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    /*
     * Block SIGQUIT and save current signal mask.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5);   /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    /*
     * Reset signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5);   /* SIGQUIT here will terminate with core file */
    exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
}
```

322

进程阻塞SIGQUIT信号，保存了当前信号屏蔽字（以便以后恢复），然后休眠5秒钟。在此期间所产生的退出信号SIGQUIT都会被阻塞，而不递送至该进程，直到该信号不再被阻塞。在5秒钟休眠结束后，检查该信号是否是未决的，然后将SIGQUIT设置为不再阻塞。

注意，在设置SIGQUIT为阻塞时，我们保存了旧屏蔽字。为了解除对该信号的阻塞，用旧屏蔽字重新设置了进程信号屏蔽字（SIG_SETMASK）。另一种方法是用SIG_UNBLOCK使阻塞的信号不再被阻塞。但是，应当了解如果编写一个可能由其他人使用的函数，而且需要在函数中阻塞一个信号，则不能用SIG_UNBLOCK简单地解除对此信号的阻塞，这是因为此函数的调用者在调用本函数之前可能也阻塞了此信号。在这种情况下必须使用SIG_SETMASK将信号屏蔽字复位为原先值，这样也就能继续阻塞该信号。10.18节的system函数部分有这样一个例子。

在休眠期间如果产生了退出信号，那么此时该信号是未决的，但是不再受阻塞，所以在sigprocmask返回之前，它被递送到调用进程。从程序的输出中可以看出：SIGQUIT处理程序（sig_quit）中的printf语句先执行，然后再执行sigprocmask之后的printf语句。

然后该进程再休眠5秒钟。如果在此期间再产生退出信号，那么因为在上次捕捉到该信号时，已将其处理方式设置为默认动作，所以这一次它就会使该进程终止。在下列输出中，当我们在终端上键入退出字符Ctrl+\时，终端打印^\（终端退出字符）：

323

```
$ ./a.out
^\                              产生信号一次（在5秒钟之内）
SIGQUIT pending                 从sleep返回后
caught SIGQUIT                  在信号处理程序中
SIGQUIT unblocked               从sigprocmask返回后
^\Quit(coredump)                再次产生信号
$ ./a.out
^\^\^\^\^\^\^\^\^\^\            产生信号10次（在5秒钟之内）
SIGQUIT pending
caught SIGQUIT                  只产生信号一次
SIGQUIT unblocked
^\Quit(coredump)                再次产生信号
```

在shell发现其子进程异常终止时，会输出"Quit (coredump)"消息。注意，第二次运行该程序时，在进程休眠期间我们使SIGQUIT信号产生了10次，但是解除了对该信号的阻塞后，只会向进程传送一次SIGQUIT。从中可以看出在此系统上没有对信号进行排队。 □

10.14 sigaction函数

sigaction函数的功能是检查或修改与指定信号相关联的处理动作（或同时执行这两种操作）。此函数取代了UNIX早期版本使用的signal函数。在本节末尾用sigaction函数实现了signal。

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);

                                    返回值：若成功则返回0，若出错则返回-1
```

其中，参数*signo*是要检测或修改其具体动作的信号编号。若*act*指针非空，则要修改其动作。如果*oact*指针非空，则系统经由*oact*指针返回该信号的上一个动作。此函数使用下列结构：

```
struct sigaction {
  void      (*sa_handler)(int);   /* addr of signal handler, */
                                  /* or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask;               /* additional signals to block */
  int       sa_flags;             /* signal options, Figure 10.16 */

  /* alternate handler */
  void      (*sa_sigaction)(int, siginfo_t *, void *);
};
```

324

当更改信号动作时，如果sa_handler字段包含一个信号捕捉函数的地址（与常量SIG_IGN或SIG_DFL相对），则sa_mask字段说明了一个信号集，在调用该信号捕捉函数之前，这一信号集要加到进程的信号屏蔽字中。仅当从信号捕捉函数返回时再将进程的信号屏蔽字复

位为原先值。这样，在调用信号处理程序时就能阻塞某些信号。在信号处理程序被调用时，操作系统建立的新信号屏蔽字包括正被递送的信号。因此保证了在处理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞到对前一个信号的处理结束为止。回忆10.8节，若同一种信号多次发生，通常并不将它们排队，所以如果在某种信号被阻塞时它发生了五次，那么对这种信号解除阻塞后，其信号处理函数通常只会被调用一次。

一旦对给定的信号设置了一个动作，那么在调用sigaction显式地改变它之前，该设置就一直有效。这种处理方式与早期的不可靠信号机制不同，而符合了POSIX.1在这方面的要求。

*act*结构的sa_flags字段指定对信号进行处理的各个选项。表10-5详细列出了这些选项的意义。若该标志已定义在基本POSIX.1标准中，那么SUS列包含•；若该标志定义在基本POSIX.1标准的XSI扩展中，那么该列包含XSI。

表10-5 处理每个信号的选项标志（sa_flags）

选 项	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说 明
SA_INTERRUPT			•			由此信号中断的系统调用不会自动重启动（针对sigaction的XSI默认处理方式）。详见10.5节
SA_NOCLDSTOP	•	•	•	•	•	若*signo*是SIGCHLD，当子进程停止时（作业控制），不产生此信号。当子进程终止时，仍旧产生此信号（但请参阅下面说明的SA_NOCLDWAIT选项）。若已设置此标志，则当停止的进程继续运行时，作为XSI扩展，不发送SIGCHLD信号
SA_NOCLDWAIT	XSI	•	•	•	•	若*signo*是SIGCHLD，则当调用进程的子进程终止时，不创建僵死进程。若调用进程在后面调用*wait*，则调用进程阻塞，直到其所有子进程都终止，此时返回-1，并将errno设置为ECHILD（见10.7节）
SA_NODEFER	XSI	•	•	•	•	当捕捉到此信号时，在执行其信号捕捉函数时，系统不自动阻塞此信号（除非sa_mask包括了此信号）。注意，此种类型的操作对应于早期的不可靠信号
SA_ONSTACK	XSI	•	•	•	•	若用sigaltstack(2)声明了一替换栈，则将此信号递送给替换栈上的进程
SA_RESETHAND	XSI	•	•	•	•	在此信号捕捉函数的入口处，将此信号的处理方式复位为SIG_DFL，并清除SA_SIGINFO标志。注意，此种类型的信号对应于早期的不可靠信号。但是，不能自动复位SIGILL和SIGTRAP这两个信号的配置。设置此标志使sigaction的行为如同SA_NODEFER标志也设置了一样
SA_RESTART	XSI	•	•	•	•	由此信号中断的系统调用会自动重启动（参见10.5节）
SA_SIGINFO	•	•	•	•	•	此选项对信号处理程序提供了附加信息：一个指向siginfo结构的指针以及一个指向进程上下文标识符的指针

sa_sigaction字段是一个替代的信号处理程序，当在sigaction结构中使用了SA_SIGINFO标志时，使用该信号处理程序。对于sa_sigaction字段和sa_handler字段这两者，其实现可能使用同一存储区，所以应用程序只能一次使用这两个字段中的一个。

通常，按下列方式调用信号处理程序：

```
void handler(int signo);
```

但是，如果设置了SA_SIGINFO标志，那么按下列方式调用信号处理程序：

```
void handler(int signo, siginfo_t *info, void *context);
```

siginfo_t结构包含了信号产生原因的有关信息。该结构的大致样式如下所示。POSIX.1依从的所有实现必须至少包括si_signo和si_code成员。另外，XSI依从的实现至少应包含下列字段：

```
struct siginfo {
  int    si_signo;  /* signal number */
  int    si_errno;  /* if nonzero, errno value from <errno.h> */
  int    si_code;   /* additional info (depends on signal) */
  pid_t  si_pid;    /* sending process ID */
  uid_t  si_uid;    /* sending process real user ID */
  void  *si_addr;   /* address that caused the fault */
  int    si_status; /* exit value or signal number */
  long   si_band;   /* band number for SIGPOLL */
  /* possibly other fields also */
};
```

325
~
326

表10-6示出了各种信号的si_code值，这些信号是由Single UNIX Specification定义的。注意，实现可定义附加的代码值。

若信号是SIGCHLD，则将设置si_pid、si_status和si_uid字段。若信号是SIGILL或SIGSEGV，则si_addr包含造成故障的根源地址，尽管该地址可能并不准确。若信号是SIGPOLL，那么si_band字段将包含STREAMS消息的优先级段（priority band），该消息产生POLL_IN、POLL_OUT或POLL_MSG事件（关于优先级段的详细讨论，请参见Rago[1993]）。si_errno字段包含错误编号，它对应于引发信号产生的条件，并由实现定义。

信号处理程序的*context*参数是无类型指针，它可被强制转换为ucntext_t结构类型，用于标识信号传递时进程的上下文。

> 当实现支持实时信号扩展时，用SA_SIGINFO标志建立的信号处理程序将导致信号可靠地排队。一些保留信号可由实时应用程序使用。如果信号由sigqueue产生，那么siginfo结构能包含应用特有的数据。我们不再进一步讨论实时扩展。详细情况请参见Gallmeister[1995]。

实例：signal函数

现在用sigaction实现signal函数。很多平台都是这样做的（POSIX.1的Rationale也说明这是POSIX所希望的）。另一方面，有些系统支持旧的不可靠信号语义signal函数。其目的是实现二进制向后兼容，除非明确要求旧的不可靠语义（为了向后兼容），否则应当使用下面的signal实现，或者直接调用sigaction（可以在调用sigaction时指定SA_RESETHAND和SA_NODEFER选项以实现旧语义的signal函数）。本书中所有调用signal的实例均调用程序清单10-12中实现的该函数。

表10-6 siginfo_t代码值

信 号	代 码	原 因
SIGILL	ILL_ILLOPC	非法操作码
	ILL_ILLOPN	非法操作数
	ILL_ILLADR	非法地址模式
	ILL_ILLTRP	非法陷入
	ILL_PRVOPC	特权操作码
	ILL_PRVREG	特权寄存器
	ILL_COPROC	协处理器出错
	ILL_BADSTK	内部栈出错
SIGFPE	FPE_INTDIV	整数除以0
	FPE_INTOVF	整数溢出
	FPE_FLTDIV	浮点除以0
	FPE_FLTOVF	浮点上溢
	FPE_FLTUND	浮点下溢
	FPE_FLTRES	浮点不精确结果
	FPE_FLTINV	无效的浮点运算
	FPE_FLTSUB	下标越界
SIGSEGV	SEGV_MAPERR	地址未映射到对象
	SEGV_ACCERR	对于映射对象的无效权限
SIGBUS	BUS_ADRALN	无效的地址对齐
	BUS_ADRERR	不存在的物理地址
	BUS_OBJERR	对象特有的硬件出错
SIGTRAP	TRAP_BRKPT	进程断点陷入
	TRAP_TRACE	进程跟踪陷入
SIGCHLD	CLD_EXITED	子进程已终止
	CLD_KILLED	子进程已异常终止（无core）
	CLD_DUMPED	子进程已异常终止（有core）
	CLD_TRAPPED	被跟踪的子进程已陷入
	CLD_STOPPED	子进程已停止
	CLD_CONTINUED	停止的子进程已继续
SIGPOLL	POLL_IN	数据可读
	POLL_OUT	数据可写
	POLL_MSG	输入消息可用
	POLL_ERR	I/O出错
	POLL_PRI	高优先级消息可用
	POLL_HUP	设备断开连接
Any	SI_USER	kill发送的信号
	SI_QUEUE	sigqueue发送的信号（实时扩展）
	SI_TIMER	timer_settime设置的计时器超时（实时扩展）
	SI_ASYNCIO	异步I/O请求完成（实时扩展）
	SI_MESGQ	一条消息到达消息队列（实时扩展）

程序清单10-12 用sigaction实现的signal函数

```
#include "apue.h"

/* Reliable version of signal(), using POSIX sigaction().  */
Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;
#endif
    } else {
#ifdef  SA_RESTART
        act.sa_flags |= SA_RESTART;
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

327 ≀ 328

注意，必须用sigemptyset函数初始化act结构的sa_mask成员。不能保证：

```
act.sa_mask = 0;
```

会做同样的事情。

对除SIGALRM以外的所有信号，我们都有意尝试设置SA_RESTART标志，于是被这些信号中断的系统调用都能重启动。不希望重启动由SIGALRM信号中断的系统调用的原因是：我们希望对I/O操作可以设置时间限制（请回忆有关程序清单10-7的讨论）。

某些早期系统（如SunOS）定义了SA_INTERRUPT标志。这些系统的默认方式是重新启动被中断的系统调用，而指定此标志则使系统调用被中断后不再重启动。Linux定义SA_INTERRUPT标志，以便与使用该标志的应用程序兼容。但是，如若信号处理程序是用sigaction设置的，那么其默认方式是不重新启动系统调用。Single UNIX Specification的XSI扩展规定，除非说明了SA_RESTART标志，否则sigaction函数不再重启动被中断的系统调用。 □

实例：signal_intr函数

程序清单10-13是signal函数的另一种版本，它力图阻止任何被中断的系统调用重启动。

程序清单10-13 signal_intr函数

```
#include "apue.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef  SA_INTERRUPT
    act.sa_flags |= SA_INTERRUPT;
```

```
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

如果系统定义了SA_INTERRUPT标志，那么为了提高可移植性，我们在sa_flags中增加该标志，这样也就阻止了被中断的系统调用重启动。 □

10.15 sigsetjmp和siglongjmp函数

7.10节说明了用于非局部转移的setjmp和longjmp函数。在信号处理程序中经常调用longjmp函数以返回到程序的主循环中，而不是从该处理程序返回。程序清单10-5和程序清单10-8中已经出现了这种情况。

329

但是，调用longjmp有一个问题。当捕捉到一个信号时，进入信号捕捉函数，此时当前信号被自动地加到进程的信号屏蔽字中。这阻止了后来产生的这种信号中断该信号处理程序。如果用longjmp跳出信号处理程序，那么，对此进程的信号屏蔽字会发生什么呢？

> 在FreeBSD 5.2.1和Mac OS X 10.3中，setjmp和longjmp保存和恢复信号屏蔽字。但是，Linux 2.4.22和Solaris 9并不执行这种操作。FreeBSD 5.2.1和Mac OS X 10.3提供函数_setjmp和_longjmp，它们也不保存和恢复信号屏蔽字。

为了允许两种形式的行为并存，POSIX.1并没有说明setjmp和longjmp对信号屏蔽字的作用，而是定义了两个新函数sigsetjmp和siglongjmp。在信号处理程序中进行非局部转移时应当使用这两个函数。

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);

                    返回值：若直接调用则返回0，若从siglongjmp调用返回则返回非0值

void siglongjmp(sigjmp_buf env, int val);
```

这两个函数与setjmp和longjmp之间的唯一区别是sigsetjmp增加了一个参数。如果*savemask*非0，则sigsetjmp在*env*中保存进程的当前信号屏蔽字。调用siglongjmp时，如果带非0 *savemask*的sigsetjmp调用已经保存了*env*，则siglongjmp从其中恢复保存的信号屏蔽字。

实　例

程序清单10-14演示了在信号处理程序被调用时，系统所设置的信号屏蔽字如何自动地包括刚被捕捉到的信号。该程序也通过实例说明了如何使用sigsetjmp和siglongjmp函数。

程序清单10-14　信号屏蔽字、sigsetjmp和siglongjmp实例

```
#include "apue.h"
#include <setjmp.h>
#include <time.h>

static void                     sig_usr1(int), sig_alrm(int);
```

```
static sigjmp_buf             jmpbuf;
static volatile sig_atomic_t  canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    pr_mask("starting main: ");

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("ending main: ");
        exit(0);
    }
    canjump = 1;    /* now sigsetjmp() is OK */

    for ( ; ; )
        pause();
}

static void
sig_usr1(int signo)
{
    time_t  starttime;

    if (canjump == 0)
        return;     /* unexpected signal, ignore */

    pr_mask("starting sig_usr1: ");
    alarm(3);               /* SIGALRM in 3 seconds */
    starttime = time(NULL);
    for ( ; ; )             /* busy wait for 5 seconds */
        if (time(NULL) > starttime + 5)
            break;
    pr_mask("finishing sig_usr1: ");

    canjump = 0;
    siglongjmp(jmpbuf, 1);  /* jump back to main, don't return */
}

static void
sig_alrm(int signo)
{
    pr_mask("in sig_alrm: ");
}
```

330

此程序演示了另一种技术，只要在信号处理程序中调用siglongjmp，就应使用这种技术。仅在调用sigsetjmp之后才将变量canjump设置为非0值。在信号处理程序中检测此变量，仅当它为非0值时才调用siglongjmp。这提供了一种保护机制，使得在jmpbuf（跳转缓冲）尚未由sigsetjmp初始化时，防止调用信号处理程序（在本程序中，调用siglongjmp之后程序很快就结束，但是在较大的程序中，在调用siglongjmp之后的一段较长时间内，信号处理程序可能仍旧被设置）。在一般的C代码中（不是信号处理程序），对于longjmp并不需要这种保护措施。但是，因为信号可能在任何时候发生，所以在信号处理程序中，需要这种保护措施。

331

在程序中使用了数据类型sig_atomic_t，这是由ISO C标准定义的变量类型，在写这种类型的变量时不会被中断。它意味着在具有虚拟存储器的系统上这种变量不会跨越页边界，可以用一条机器指令对其进行访问。这种类型的变量总是包括ISO类型修饰符volatile，其原

因是：该变量将由两个不同的控制线程——main函数和异步执行的信号处理程序访问。

图10-1显示了此程序的执行时间顺序。

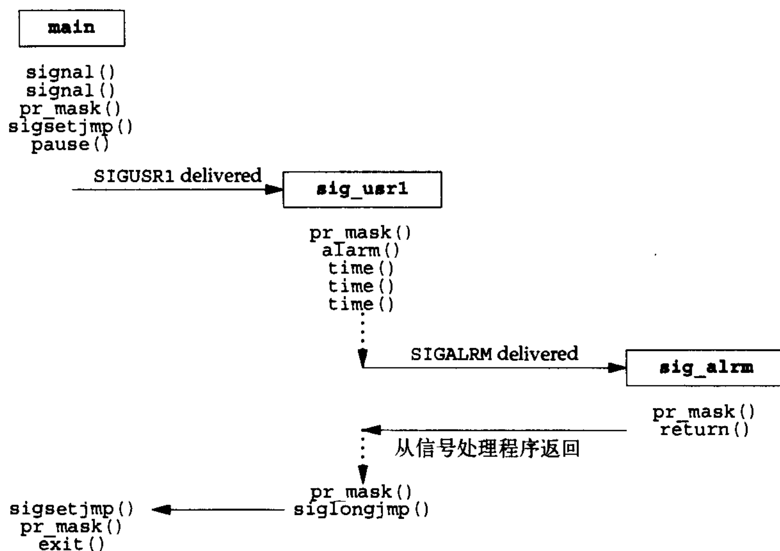


图10-1　处理两个信号的实例程序的时间顺序

可将图10-1分成三部分：左面部分（对应于main）、中间部分（sig_usr1）和右面部分（sig_alrm）。在进程执行左面部分时，信号屏蔽字是0（没有信号是阻塞的）。而执行中间部分时，其信号屏蔽字是SIGUSR1。执行右面部分时，信号屏蔽字是SIGUSR1 | SIGALRM。

执行程序清单10-14，得到下面的输出：

```
$ ./a.out &                          在后台启动进程
starting main:
[1]   531                            作业控制shell打印其进程ID
$ kill -USR1 531                     向该进程发送SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                                     键入回车
[1] +  Done          ./a.out &
```

332

该输出与我们所期望的相同：当调用一个信号处理程序时，被捕捉到的信号加到进程的当前信号屏蔽字中。当从信号处理程序返回时，恢复原来的屏蔽字。另外，siglongjmp恢复了由sigsetjmp保存的信号屏蔽字。

如果在Linux 2.4.22中将程序清单10-14中的sigsetjmp和siglongjmp分别替换成_setjmp和_longjmp（在FreeBSD中，则替换成_setjmp和_longjmp），则最后一行输出变成：

```
ending main: SIGUSR1
```

这意味着在调用_setjmp之后执行main函数时，其SIGUSR1是阻塞的。这多半不是我们所希望的。□

10.16　sigsuspend函数

上面已经说明，更改进程的信号屏蔽字可以阻塞所选择的信号，或解除对它们的阻塞。使

用这种技术可以保护不希望由信号中断的代码临界区。如果希望对一个信号解除阻塞，然后pause以等待以前被阻塞的信号发生，则又将如何呢？假定信号是SIGINT，实现这一点的一种不正确的方法是：

```
sigset_t    newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

/* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region of code */

/* reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

/* window is open */
pause();  /* wait for signal to occur */

/* continue processing */
```

如果在信号阻塞时将其发送给进程，那么该信号的传递就被推迟直到对它解除了阻塞。对应用程序而言，该信号好像发生在解除对SIGINT的阻塞和pause之间（取决于内核如何实现信号。）。如果发生了这种情况，或者如果在解除阻塞时刻和pause之间确实发生了信号，那么就产生了问题。因为我们可能不会再见到该信号，所以从这种意义上而言，在此时间窗口中发生的信号丢失了，这样就使得pause永远阻塞。这是早期的不可靠信号机制的另一个问题。

为了纠正此问题，需要在一个原子操作中先恢复信号屏蔽字，然后使进程休眠。这种功能是由sigsuspend函数提供的。 333

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

返回值：-1，并将errno设置为EINTR

将进程的信号屏蔽字设置为由*sigmask*指向的值。在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则*sigsuspend*返回，并且将该进程的信号屏蔽字设置为调用*sigsuspend*之前的值。

注意，此函数没有成功返回值。如果它返回到调用者，则总是返回-1，并将errno设置为EINTR（表示一个被中断的系统调用）。

实例

程序清单10-15显示了保护临界区，使其不被特定信号中断的正确方法。

程序清单10-15 保护临界区不被信号中断

```
#include "apue.h"

static void sig_int(int);

int
main(void)
{
```

```
    sigset_t    newmask, oldmask, waitmask;

    pr_mask("program start: ");

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    sigemptyset(&waitmask);
    sigaddset(&waitmask, SIGUSR1);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /*
     * Block SIGINT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    /*
     * Critical region of code.
     */
    pr_mask("in critical region: ");

    /*
     * Pause, allowing all signals except SIGUSR1.
     */
    if (sigsuspend(&waitmask) != -1)
        err_sys("sigsuspend error");

    pr_mask("after return from sigsuspend: ");

    /*
     * Reset signal mask which unblocks SIGINT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    /*
     * And continue processing ...
     */
    pr_mask("program exit: ");

    exit(0);
}

static void
sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
}
```

334

注意，当sigsuspend返回时，它将信号屏蔽字设置为调用它之前的值。在本例中，SIGINT信号将被阻塞。因此将信号屏蔽字复位为早先保存的值（oldmask）。

运行程序清单10-15得到下面的输出：

```
$ ./a.out
program start:
in critical region: SIGINT
^?                                  键入中断字符
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:
```

在调用sigsuspend时，将SIGUSR1信号加到了进程信号屏蔽字中，所以当运行该信号处理程序时，我们得知信号屏蔽字已经改变了。从中可见，在sigsuspend返回时，它将信号屏蔽字恢复为调用它之前的值。 □

sigsuspend的另一种应用是等待一个信号处理程序设置一个全局变量。程序清单10-16用于捕捉中断信号和退出信号，但是希望仅当捕捉到退出信号时，才唤醒主例程。 335

程序清单10-16 用sigsuspend等待一个全局变量被设置

```
#include "apue.h"

volatile sig_atomic_t   quitflag;   /* set nonzero by signal handler */

static void
sig_int(int signo)  /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1;   /* set flag for main loop */
}

int
main(void)
{
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /*
     * Block SIGQUIT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /*
     * SIGQUIT has been caught and is now blocked; do whatever.
     */
    quitflag = 0;

    /*
     * Reset signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    exit(0);
}
```

336

此程序的示例输出是：

```
$ ./a.out
^?                    键入中断字符
interrupt
^?                    再次键入中断字符
interrupt
^?                    再一次
interrupt
^?                    再一次
interrupt
^?                    再一次
interrupt
^?                    再一次
interrupt
^?                    再一次
interrupt
^\ $                  用退出符终止
```

□

考虑到支持ISO C的非POSIX系统与POSIX系统两者之间的可移植性，在一个信号处理程序中我们唯一应当做的是赋一个值给类型为sig_atomic_t的变量。POSIX.1规定得更多一些，它详细说明了在一个信号处理程序中可以安全地调用的函数列表（见表10-3），但是如果这样编写代码，则它们可能不会正确地在非POSIX系统上运行。

实 例

可以用信号实现父、子进程之间的同步，这是信号应用的另一个实例。程序清单10-17包含了8.9节中提到的五个例程的实现，它们是：TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT和WAIT_CHILD。

程序清单10-17 父子进程可用来实现同步的例程

```
#include "apue.h"

static volatile sig_atomic_t sigflag; /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo)  /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /*
     * Block SIGUSR1 and SIGUSR2, and save current signal mask.
```

337

```
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2);     /* tell parent we're done */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for parent */
    sigflag = 0;

    /*
     * Reset signal mask to original value.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}

void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1);         /* tell child we're done */
}

void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for child */
    sigflag = 0;

    /*
     * Reset signal mask to original value.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

338

其中使用了两个用户定义的信号：SIGUSR1由父进程发送给子进程，SIGUSR2由子进程发送给父进程。程序清单15-3示出了使用管道的这五个函数的另一种实现。 □

如果在等待信号发生时希望去休眠，则使用sigsuspend函数是非常适当的（正如前面两个例子中所示），但是如果在等待信号期间希望调用其他系统函数，那么将会怎样呢？不幸的是，在单线程环境下对此问题没有妥善的解决方法。如果可以使用多线程，则可专门安排一个线程处理信号（见12.8节中的讨论）。

如果不使用线程，那么我们能尽力做到最好的是，当信号发生时，在信号捕捉程序中对一个全局变量置1。例如，若我们捕捉SIGINT和SIGALRM这两种信号，并用signal_intr函数设置这两个信号的处理程序，使得它们中断任一被阻塞的低速系统调用。当进程阻塞在select函数调用（见12.5.1节）等待低速设备输入时，很可能发生这两种信号（如果设置闹钟以阻止永远等待输入，那么对于SIGALRM信号，这是特别真实的）。处理这种问题的代码类似

于下面所示：

```
if (intr_flag)      /* flag set by our SIGINT handler */
    handle_intr();
if (alrm_flag)      /* flag set by our SIGALRM handler */
    handle_alrm();

/* signals occurring in here are lost */

while (select( ... ) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
            handle_intr();
    } else {
        /* some other error */
    }
}
```

在调用select之前测试各全局标志，如果select返回一个中断的系统调用错误，则再次进行测试。如果在前两个if语句和后随的select调用之间捕捉到两个信号中的任意一个，则问题就发生了。正如代码中的注释所指出的，在此处发生的信号丢失了。调用信号处理程序，它们设置了相应的全局变量，但是select决不会返回（除非某些数据已准备好可读）。

我们希望按顺序执行下列操作步骤：

(1) 阻塞SIGINT和SIGALRM。

339

(2) 测试两个全局变量以判别是否发生了一个信号，如果已发生则对此进行处理。

(3) 调用select（或任何其他系统调用，例如read）并解除对这两个信号的阻塞，这两个操作应当是一个原子操作。

仅当第(3)步是pause操作时，sigsuspend函数才能帮助我们。

10.17 abort函数

前面已提及abort函数的功能是使异常程序终止。

```
#include <stdlib.h>

void abort(void);
```

此函数不返回

此函数将SIGABRT信号发送给调用进程（进程不应忽略此信号）。ISO C规定，调用abort将向主机环境递送一个未成功终止的通知，其方法是调用raise (SIGABRT) 函数。

ISO C要求若捕捉到此信号而且相应信号处理程序返回，abort仍不会返回到其调用者。如果捕捉到此信号，则信号处理程序不能返回的唯一方法是它调用exit、_exit、_Exit、longjmp或siglongjmp。（10.15节讨论了longjmp和siglongjmp之间的区别。）POSIX.1也说明abort并不理会进程对此信号的阻塞和忽略。

让进程捕捉SIGABRT的意图是：在进程终止之前由其执行所需的清理操作。如果进程并不在信号处理程序中终止自己，POSIX.1声明当信号处理程序返回时，abort终止该进程。

ISO C针对此函数的规范将下列问题留由实现决定：是否要冲洗输出流以及是否要删除临

时文件（见5.13节)? POSIX.1的要求则更进一步，它要求如果abort调用终止进程，则它对所有打开标准I/O流的效果应当与进程终止前对每个流调用fclose相同。

在系统V的早期版本中，abort函数产生SIGIOT信号。更进一步讲，进程忽略此信号或者捕捉它并从信号处理程序返回，这都是可能的，在返回情况下，abort返回到它的调用者。

4.3BSD产生SIGILL信号。在此之前，该函数解除对此信号的阻塞，将其配置恢复为SIG_DFL（终止并构造core文件)。这阻止一个进程忽略或捕捉此信号。

历史上，abort的各种实现在如何处理标准I/O流方面并不相同。对于保护性的程序设计以及为提高可移植性，如果希望冲洗标准I/O流，则在调用abort之前要执行这种操作。在err_dump函数中实现了这一点（见附录B)。

因为大多数UNIX tmpfile（临时文件）的实现在创建该文件之后会立即调用unlink，所以ISO C关于临时文件的警告通常与我们无关。

340

实 例

程序清单10-18中的abort函数是按POSIX.1说明实现的。

程序清单10-18 abort的POSIX.1实现

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
abort(void)         /* POSIX-style abort() function */
{
    sigset_t            mask;
    struct sigaction    action;

    /*
     * Caller can't ignore SIGABRT, if so reset to default.
     */
    sigaction(SIGABRT, NULL, &action);
    if (action.sa_handler == SIG_IGN) {
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);
    }
    if (action.sa_handler == SIG_DFL)
        fflush(NULL);           /* flush all open stdio streams */

    /*
     * Caller can't block SIGABRT; make sure it's unblocked.
     */
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT);  /* mask has only SIGABRT turned off */
    sigprocmask(SIG_SETMASK, &mask, NULL);
    kill(getpid(), SIGABRT);    /* send the signal */

    /*
     * If we're here, process caught SIGABRT and returned.
     */
    fflush(NULL);               /* flush all open stdio streams */
    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);  /* reset to default */
```

```
    sigprocmask(SIG_SETMASK, &mask, NULL);  /* just in case ... */
    kill(getpid(), SIGABRT);                /* and one more time */
    exit(1);    /* this should never be executed ... */
}
```

首先查看是否将执行默认动作，若是则冲洗所有标准I/O流。这并不等价于对所有打开的流调用fclose（因为只冲洗，并不关闭它们），但是当进程终止时，系统会关闭所有打开的文件。如果进程捕捉此信号并返回，那么因为进程可能产生了更多的输出，所以再一次冲洗所有的流。不进行冲洗处理的唯一条件是如果进程捕捉此信号，然后调用_exit或_Exit。在这种情况下，内存中任何未冲洗的标准I/O缓冲区都被丢弃。我们假定捕捉此信号，而且_exit或_Exit的调用者并不想要冲洗缓冲区。

341

回忆10.9节，如果调用kill使其为调用者产生信号，并且如果该信号是不被阻塞的（程序清单10-18保证做到这一点），则在kill返回前该信号（或某个未决、未阻塞的信号）就被传送给了该进程。我们阻塞除SIGABRT之外的所有信号，这样就可知如果对kill的调用返回了，则该进程一定已捕捉到该信号，并且也从该信号处理程序返回。□

10.18　system函数

8.13节中已经有了一个system函数的实现，但是该版本并不执行任何信号处理。POSIX.1要求system忽略SIGINT和SIGQUIT，阻塞SIGCHLD。在给出一个正确地处理这些信号的一个版本之前，先说明为什么要考虑信号处理。

程序清单10-19使用8.13节中的system版本，用其调用ed(1)编辑器。（ed很久以来就是UNIX的组成部分。在这里使用它的原因是：它是捕捉中断和退出信号的交互式程序。若从shell调用ed，并键入中断字符，则它捕捉中断信号并打印问号。它还将对退出符的处理方式设置为忽略。）

程序清单10-19　用system调用ed编辑器

```
#include "apue.h"

static void
sig_int(int signo)
{
    printf("caught SIGINT\n");
}

static void
sig_chld(int signo)
{
    printf("caught SIGCHLD\n");
}

int
main(void)
{
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
```

```
        err_sys("signal(SIGCHLD) error");
    if (system("/bin/ed") < 0)
        err_sys("system() error");
    exit(0);
}
```

程序清单10-19用于捕捉SIGINT和SIGCHLD信号。若调用它则可得：

```
$ ./a.out
a                              将正文添加至编辑器缓冲区
Here is one line of text
.                              行首的点停止添加方式
1,$p                           打印缓冲区中的第1行至最后1行，以便观察其内容
Here is one line of text
w temp.foo                     将缓冲区写至一文件
25                             编辑器称写了25个字节
q                              离开编辑器
caught SIGCHLD
```

当编辑器终止时，系统向父进程（a.out进程）发送SIGCHLD信号。父进程捕捉它，然后从信号处理程序返回。但是若父进程正在捕捉SIGCHLD信号（因为它创建了子进程，所以应当这样做以便了解它的子进程在何时终止），那么正在执行system函数时，应当阻塞对父进程递送SIGCHLD信号。实际上，这就是POSIX.1所说明的。否则，当system创建的子进程结束时，system的调用者可能错误地认为，它自己的一个子进程结束了。于是，调用者将会调用一种wait函数以获得子进程的终止状态，这样就阻止了system函数获得子进程的终止状态，并将其作为它的返回值。

如果再次执行该程序，在这次运行时将一个中断信号传送给编辑器，则可得：

```
$ ./a.out
a                        将正文添加至编辑器缓冲区
hello, world
.                        行首的点停止添加方式
1,$p                     打印缓冲区中的第1行至最后1行，以便观察其内容
hello, world
w temp.foo               将缓冲区写至一文件
13                       编辑器称写了13个字节
^?                       键入中断符
?                        编辑器捕捉信号，打印问号
caught SIGINT            父进程执行同一操作
q                        离开编辑器
caught SIGCHLD
```

回忆9.6节可知，键入中断字符可使中断信号传送给前台进程组中的所有进程。图10-2显示了编辑程序正在运行时的各个进程的关系。

342
~
343

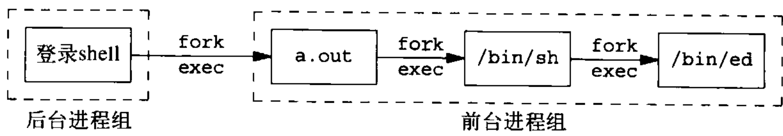


图10-2 程序清单10-19运行时的前台和后台进程组

在这一实例中，SIGINT被送给三个前台进程（shell进程忽略此信号）。从输出中可见，a.out进程和ed进程捕捉该信号。但是，当用system运行另一个程序（例如ed）时，不应使父、子进程两者都捕捉终端产生的两个信号：中断和退出。这两个信号只应送给正在运行的程

序：子进程。因为由system执行的命令可能是交互式命令（如本例中的ed程序），以及因为system的调用者在程序执行时放弃了控制，等待该执行程序的结束，所以system的调用者就不应接收这两个终端产生的信号。这就是为什么POSIX.1规定system的调用者应当忽略这两个信号的原因。 □

程序清单10-20显示了system函数的另一个实现，它进行了所要求的信号处理。

程序清单10-20　system函数的POSIX.1正确实现

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <signal.h>
#include    <unistd.h>

int
system(const char *cmdstring)   /* with appropriate signal handling */
{
    pid_t               pid;
    int                 status;
    struct sigaction    ignore, saveintr, savequit;
    sigset_t            chldmask, savemask;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    ignore.sa_handler = SIG_IGN;    /* ignore SIGINT and SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);
    sigemptyset(&chldmask);         /* now block SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {          /* child */
        /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* exec error */
    } else {                        /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

    /* restore previous signal actions & reset signal mask */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
```

344

```
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);

    return(status);
}
```

如果链接程序清单10-19与system函数的这一实现，那么所产生的二进制代码与上一个有缺陷的程序相比较，存在如下差别：

(1) 当我们键入中断或退出字符时，不向调用进程发送信号。

(2) 当ed命令终止时，不向调用进程发送SIGCHLD信号。作为替代，在程序末尾的sigprocmask调用对SIGCHLD信号解除阻塞之前，SIGCHLD信号一直被阻塞。而对sigprocmask函数的这一次调用是在system函数调用waitpid取到了子进程的终止状态之后。

POSIX.1指出，在SIGCHLD未决期间，如若wait或waitpid返回了子进程的状态，那么SIGCHLD信号不应递送给该父进程，除非另一个子进程的状态也可用。本书讨论的四种实现都没有实现这种语义。作为替代，在system函数调用了waitpid后，SIGCHLD保持为未决；当解除了对此信号的阻塞后，它被递送至调用者。如果我们在程序清单10-19的sig_chld函数中调用wait，则它将返回−1，并将errno设置为ECHILD，因为system函数已取到子进程的终止状态。

345

很多较早的书籍中使用下列程序段，它忽略中断和退出信号：

```
if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {
    /* child */
    execl(...);
    _exit(127);
}

/* parent */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);
```

这段代码的问题是：在fork之后不能保证父进程还是子进程先运行。如果子进程先运行，父进程在一段时间后再运行，那么在父进程将中断信号的配置更改为忽略之前，就可能产生这种信号。由于这种原因，程序清单10-20在fork之前就改变对该信号的配置。

注意，子进程在调用execl之前要先恢复这两个信号的配置。如同8.10节中所说明的一样，这就允许在调用者配置的基础上，execl可将它们的配置更改为默认值。 □

system的返回值

注意system的返回值，它是shell的终止状态，但shell的终止状态并不总是执行命令字符串进程的终止状态。程序清单8-13中有一些例子，其结果正是我们所期望的。如果执行一条如date那样的简单命令，其终止状态是0。执行shell命令exit 44，则得终止状态44。在信号方面又如何呢？

运行程序清单8-14，并向正在执行的命令发送一些信号：

```
$ tsys "sleep 30"
^?normal termination, exit status = 130    键入中断符
$ tsys "sleep 30"
^\sh: 946 Quit                             键入退出符
normal termination, exit status = 131
```

当用中断信号终止sleep时，pr_exit函数（见程序清单8-3）认为它正常终止。当用退出键杀死sleep进程时，会发生同样的事情。终止状态130、131又是怎样得到的呢？原来Bourne shell有一个在其文档中没有说清楚的特性，其终止状态是128加上一个信号编号，该信号终止了正在执行的命令。用交互方式使用shell可以看到这一点。

346

```
$ sh                                  确保运行Bourne shell
$ sh -c "sleep 30"
^?                                    键入中断符
$ echo $?                             打印最后一条命令的终止状态
130
$ sh -c "sleep 30"
^\sh: 962 Quit - core dumped          键入退出符
$ echo $?                             打印最后一条命令的终止状态
131
$ exit                                离开Bourne shell
```

在所使用的系统中，SIGINT的值为2，SIGQUIT的值为3，于是给出shell终止状态130、131。

再试一个类似的例子，这一次将一个信号直接送给shell，然后观察system返回什么：

```
$ tsys "sleep 30" &                      这一次在后台启动它
9257
$ ps -f                                  查看进程ID
     UID   PID  PPID  TTY      TIME CMD
     sar  9260   949  pts/5    0:00 ps -f
     sar  9258  9257  pts/5    0:00 sh -c sleep 60
     sar   949   947  pts/5    0:01 /bin/sh
     sar  9257   949  pts/5    0:00 tsys sleep 60
     sar  9259  9258  pts/5    0:00 sleep 60
$ kill -KILL 9258                        杀死shell自身
abnormal termination, signal number = 9
```

从中可见仅当shell本身异常终止时，system的返回值才报告一个异常终止。

在编写使用system函数的程序时，一定要正确地解释返回值。如果直接调用fork、exec和wait，则终止状态与调用system是不同的。

10.19 sleep函数

在本书的很多例子中都已使用了sheep函数，在程序清单10-4和程序清单10-5中有sleep的两个有缺陷的实现。

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);

                                          返回值：0或未休眠够的秒数
```

此函数使调用进程被挂起，直到满足以下条件之一：

(1) 已经过了*seconds*所指定的墙上时钟时间。

347

(2) 调用进程捕捉到一个信号并从信号处理程序返回。

如同alarm信号一样，由于其他系统活动，实际返回时间比所要求的会迟一些。

在第1种情形中，返回值是0。当由于捕捉到某个信号sleep提早返回时（第2种情形），返回值是未睡够的秒数（所要求的时间减去实际休眠时间）。

尽管sleep可以用alarm函数（见10.10节）实现，但这并不是必需的。如果使用alarm，则这两个函数之间可能交互作用。POSIX.1标准对这些交互作用并未做任何说明。例如，若先调用alarm(10)，过了3秒后又调用sleep(5)，那么将如何呢？sleep将在5秒后返回（假定在这段时间内没有捕捉到另一个信号），但是否在2秒后又产生另一个SIGALRM信号呢？这些细节依赖于实现。

Solaris 9用alarm实现sleep。Solaris 9 sleep(3)手册页中说明以前安排的闹钟仍被正常处理。例如，在前面的例子中，在sleep返回之前，它安排在2秒后再次到达闹钟时间。在这种情况下，sleep返回0（显然，sleep必须保存SIGALRM信号处理程序的地址，并在返回前复位它）。另外，如果先做一次alarm(6)，3秒钟之后又做一次sleep(5)，则在3秒后sleep返回（第一次闹钟时间已到），而不是5秒钟。此时，sleep的返回值是未睡够的时间2秒。

FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3则使用另一种技术：用nanosleep(2)提供时间延迟。该函数由Single UNIX Specification的实时扩展说明，它提供的时间延迟是高分辨率的。该函数可以使sleep的实现与信号无关。

考虑到可移植性，不应对sleep的实现作任何假定，但是如果混合调用sleep和其他与计时有关的函数，则需了解它们之间可能产生的交互作用。

实例

程序清单10-21是一个POSIX.1 sleep函数的实现。此函数是程序清单10-4的修改版，它可靠地处理信号，避免了早期实现中的竞争条件，但是仍未处理与以前设置的闹钟的交互作用（正如前面提到的，POSIX.1并未显式地定义这些交互作用）。

程序清单10-21 sleep的可靠实现

```
#include "apue.h"

static void
sig_alrm(int signo)
{
    /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int
sleep(unsigned int nsecs)
{
    struct sigaction    newact, oldact;
    sigset_t            newmask, oldmask, suspmask;
    unsigned int        unslept;

    /* set our handler, save previous information */
    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);

    /* block SIGALRM and save current signal mask */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
```

348

```
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(nsecs);

    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);    /* make sure SIGALRM isn't blocked */
    sigsuspend(&suspmask);            /* wait for any signal to be caught */

    /* some signal has been caught, SIGALRM is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);  /* reset previous action */

    /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return(unslept);
}
```

与程序清单10-4相比，为了可靠地实现sleep，程序清单10-21的代码比较长。程序中没有使用任何形式的非局部转移（如程序清单10-5为了避免在alarm和pause之间的竞争条件所做的那样），所以对处理SIGALRM信号期间可能执行的其他信号处理程序没有任何影响。□

10.20 作业控制信号

在表10-1所示的信号中，POSIX.1认为有6个与作业控制有关：

SIGCHLD　子进程已停止或终止。

SIGCONT　如果进程已停止，则使其继续运行。

SIGSTOP　停止信号（不能被捕捉或忽略)。

SIGTSTP　交互式停止信号。

SIGTTIN　后台进程组成员读控制终端。

349

SIGTTOU　后台进程组成员写到控制终端。

除SIGCHLD以外，大多数应用程序并不处理这些信号：交互式shell则通常做处理这些信号的所有工作。当键入挂起字符（通常是Ctrl+Z）时，SIGTSTP被送至前台进程组的所有进程。当我们通知shell在前台或后台恢复运行一个作业时，shell向该作业中的所有进程发送SIGCONT信号。与此类似，如果向一个进程递送了SIGTTIN或SIGTTOU信号，则根据系统默认的方式停止此进程，作业控制shell了解到这一点后就通知我们。

一个例外是管理终端的进程——例如，vi(1)编辑器。当用户要挂起它时，它需要能了解到这一点，这样就能将终端状态恢复到vi启动时的情况。另外，当在前台恢复它时，它需要将终端状态设置回它所希望的状态，并需要重新绘制终端屏幕。可以在下面的例子中观察到与vi类似的程序是如何处理这种情况的。

在作业控制信号间有某种交互作用。当对一个进程产生四种停止信号（SIGTSTP、SIGSTOP、SIGTTIN或SIGTTOU）中的任意一种时，对该进程的任一未决SIGCONT信号就会被丢弃。与此类似，当对一个进程产生SIGCONT信号时，对同一进程的任一未决停止信号将被丢弃。

注意，如果进程是停止的，SIGCONT的默认动作是继续运行该进程，否则忽略此信号。通常，对该信号无需做任何事情。当对一个停止的进程产生一个SIGCONT信号时，该进程就继续运行，即使该信号是被阻塞或忽略的也是如此。

程序清单10-22例示了当一个程序处理作业控制时所使用的规范代码序列。该程序只是将其标准输入复制到其标准输出，而在信号处理程序中以注释形式给出了管理屏幕的程序所执行的典型操作。当程序清单10-22启动时，仅当SIGTSTP信号的配置是SIG_DFL，它才安排捕捉该信号。其理由是：当此程序由不支持作业控制的shell（例如/bin/sh）启动时，此信号的配置应当设置为SIG_IGN。实际上，shell并不显式地忽略此信号，而是由init将这三个作业控制信号SIGTSTP、SIGTTIN和SIGTTOU设置为SIG_IGN。然后，这种配置由所有登录shell继承。只有作业控制shell才应将这三个信号复位为SIG_DFL。

程序清单10-22 如何处理SIGTSTP

```
#include "apue.h"

#define BUFFSIZE    1024

static void sig_tstp(int);

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    /*
     * Only catch SIGTSTP if we're running with a job-control shell.
     */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}

static void
sig_tstp(int signo) /* signal handler for SIGTSTP */
{
    sigset_t    mask;

    /* ... move cursor to lower left corner, reset tty mode ... */

    /*
     * Unblock SIGTSTP, since it's blocked while we're handling it.
     */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL);   /* reset disposition to default */

    kill(getpid(), SIGTSTP);    /* and send the signal to ourself */

    /* we won't return from the kill until we're continued */

    signal(SIGTSTP, sig_tstp);  /* reestablish signal handler */
```

```
    /* ... reset tty mode, redraw screen ... */
}
```

当键入挂起字符时，进程接到SIGTSTP信号，然后调用该信号处理程序。此时，应当进行与终端有关的处理：将光标移到左下角，恢复终端工作方式等等。在将SIGTSTP复位为默认值（停止该进程），并且解除了对此信号的阻塞之后，进程向自己发送同一信号SIGTSTP。因为正在处理SIGTSTP信号，而在捕捉该信号期间系统会自动阻塞它，所以应当解除对此信号的阻塞。到达这一点时，系统停止该进程。仅当某个进程（通常是正响应一个交互式fg命令的作业控制shell）向该进程发送一个SIGCONT信号时，该进程才继续。我们不捕捉SIGCONT信号。该信号的默认配置是继续运行停止的进程，当此发生时，此程序如同从kill函数返回一样继续运行。当此程序继续运行时，将SIGTSTP信号复位为捕捉，并且做我们所希望做的终端处理（例如重新绘制屏幕）。 □

350~351

10.21 其他特征

本节介绍某些依赖于实现的信号的其他特征。

1. 信号名字

某些系统提供数组

```
extern char *sys_siglist[];
```

数组下标是信号编号，给出一个指向信号字符串名字的指针。

FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3都提供这种信号名数组。Solaris 9也提供信号名数组，但该数组名是_sys_siglist。

这些系统通常也提供函数psignal。

```
#include <signal.h>

void psignal(int signo, const char *msg);
```

字符串*msg*（通常是程序名）输出到标准出错文件，后接一个冒号和一个空格，再接着对该信号的说明，最后是一个换行符。该函数类似于perror（1.7节）。

另一个常用函数是strsignal。它类似于strerror（也见1.7节）。

```
#include <string.h>

char *strsignal(int signo);

                    返回值：指向描述该信号的字符串的指针
```

给出一个信号编号，strsignal将返回说明该信号的字符串。应用程序可用该字符串打印关于接收到信号的出错消息。

本书讨论的所有平台都提供psignal和strsignal函数，但相互之间有些差别。在Solaris 9中，若信号编号无效，那么，strsignal将返回一个空指针，而FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3则返回一个字符串，它指出信号编号是不可识别的。另外，在Solaris中，为了得到psignal的函数原型，需在程序中包括<siginfo.h>。

352

2. 信号映射

Solaris提供一对函数，一个函数将信号编号映射为信号名，另一个则反之。

```
#include <signal.h>

int sig2str(int signo, char *str);

int str2sig(const char *str, int *signop);
```

两个函数的返回值：若成功则返回0，若出错则返回−1

在编写交互式程序，其中需接收和打印信号名和编号时，这两个函数是有用的。

sig2str函数将给定信号编号翻译成字符串，并将结果存放在*str*指向的存储区。调用者必须保证该存储区足够大，可以保存最长的字符串，包括终止null字节。Solaris在<signal.h>中包含了常量SIG2STR_MAX，它定义了最大字符串长度。该字符串包括不带“SIG”前缀的信号名。例如，SIGKILL被翻译为字符串“KILL”，并存放在*str*指向的存储缓冲区中。

str2sig函数将给出的名字翻译成信号编号。该信号编号存放在*signop*指向的整型中。名字要么是不带“SIG”前缀的信号名，要么是表示十进制信号编号的字符串（例如“9”）。

注意，sig2str和str2sig偏离了一般实践，当它们失败时，并不设置errno。

10.22 小结

信号用于大多数复杂的应用程序中。理解进行信号处理的原因和方式对于高级UNIX编程极其重要。本章对UNIX信号进行了详细而且比较深入的介绍。首先说明了早期信号实施的问题以及它们是如何显现出来的。然后介绍了POSIX.1的可靠信号概念以及所有相关的函数。在此基础上接着提供了abort、system和sleep函数的POSIX.1实现。最后以观察分析作业控制信号以及信号名和信号编号之间的转换结束。

习题

10.1 删除程序清单10-1中的for(;;)语句，结果会怎样？为什么？

10.2 实现10-21节中说明的sig2str函数。　353

10.3 画出运行程序清单10-6时的栈帧情况。

10.4 程序清单10-8中利用setjmp和longjmp设置I/O操作的超时，下面的代码也常用于此种目的：

```
signal(SIGALRM, sig_alrm);
alarm(60);
if (setjmp(env_alrm) != 0) {
    /* handle timeout */
    ...
}
...
```

这段代码有什么错误？

10.5 仅使用一个计时器（alarm或较高精度的setitimer），构造一组函数，使得进程在该单一计时器基础上可以设置任意数量的计时器。

10.6 编写一段程序测试程序清单10-17中父进程和子进程的同步函数，要求进程创建一个文件并向文件写一个整数0，然后进程调用fork，接着父进程和子进程交替增加文件中的计

数器值，每次计数器值增1时，打印是哪一个进程（子进程或父进程）进行了该增1操作。

10.7 在程序清单10-18所示的函数中，若调用者捕捉了SIGABRT并从该信号处理程序中返回，为什么不是仅仅调用_exit，而要复位其默认设置并再次调用kill？

10.8 为什么在siginfo结构（见10.14节）的si_uid字段中包括实际用户ID而非有效用户ID？

10.9 重写程序清单10-10中的函数，要求它处理表10-1中的所有信号量，每次循环处理当前信号屏蔽字中的一个信号量（并不是对每一个可能的信号量都循环一次）。

10.10 编写一段程序，要求在一个无限循环中调用sleep(60)函数，每5分钟（即5次循环）取当前的日期和时间，并打印tm_sec字段。将程序执行一晚上，请解释其结果。有些程序（如BSD中的cron精灵进程）每分钟运行一次，它是如何处理这类工作的？

10.11 修改程序清单3-3，要求：(a) 将BUFFSIZE改为100；(b) 用signal_intr函数捕捉SIGXFSZ信号量并打印消息，然后从信号量处理程序中返回；(c) 如果没有写满请求的字节数，则打印write的返回值。将软资源限制RLIMIT_FSIZE（见7.11节）变为1 024字节（在shell中设置软资源限制，如果不行就直接在程序中调用setrlimit），然后复制一个大于1 024字节的文件，在各种不同的系统上运行新程序，其结果如何？为什么？

10.12 编写一段调用fwrite的程序，它使用一个较大的缓冲区（几百兆），在调用fwrite前调用alarm使得一秒钟以后产生信号。在信号处理程序中打印捕捉到的信号，然后返回。调用fwrite可以完成吗？结果如何？

354

第11章 线程

11.1 引言

在前面的章节中讨论了进程，学习了UNIX进程的环境、进程间的关系以及控制进程的不同方式。可以看出在相关的进程间能够存在一定的共享。

本章将进一步深入考察进程，了解如何使用多个*控制线程*（或简称为*线程*）在单进程环境中执行多个任务。一个进程中的所有线程都可以访问该进程的组成部件，如文件描述符和内存。

无论何时，只要单个资源需要在多个用户间共享，就必须处理一致性问题。本章的最后将讨论目前可用的同步机制，该机制用以防止多个线程查看到不一致的共享资源。

11.2 线程概念

典型的UNIX进程可以看成只有一个控制线程：一个进程在同一时刻只做一件事情。有了多个控制线程以后，在程序设计时可以把进程设计成在同一时刻能够做不止一件事，每个线程处理各自独立的任务。这种方法有很多好处。

- 通过为每种事件类型的处理分配单独的线程，能够简化处理异步事件的代码。每个线程在进行事件处理时可以采用同步编程模式，同步编程模式要比异步编程模式简单得多。 355
- 多个进程必须使用操作系统提供的复杂机制才能实现内存和文件描述符的共享，这方面的内容将在第15章和第17章中学习。而多个线程自动地可以访问相同的存储地址空间和文件描述符。
- 有些问题可以通过将其分解从而改善整个程序的吞吐量。在只有一个控制线程的情况下，单个进程需要完成多个任务时，实际上需要把这些任务串行化；有了多个控制线程，相互独立的任务的处理就可以交叉进行，只需要为每个任务分配一个单独的线程，当然只有在处理过程互不依赖的情况下，两个任务的执行才可以穿插进行。
- 交互的程序同样可以通过使用多线程实现响应时间的改善，多线程可以把程序中处理用户输入输出的部分与其他部分分开。

有些人把多线程的程序设计与多处理器系统联系起来，但是即使程序运行在单处理器上，也能得到多线程编程模型的好处。处理器的数量并不影响程序结构，所以不管处理器的个数是多少，程序可以通过使用线程得以简化。而且，即使多线程程序在串行化任务时不得不阻塞，由于某些线程在阻塞的时候还有另外一些线程可以运行，所以多线程程序在单处理器上运行仍然能够改善响应时间和吞吐量。

线程包含了表示进程内执行环境必需的信息，其中包括进程中标识线程的*线程ID*、一组寄存器值、栈、调度优先级和策略、信号屏蔽字、errno变量（见1.7节）以及线程私有数据（见

12.6节)。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本、程序的全局内存和堆内存、栈以及文件描述符。

我们将要讨论的线程接口来自POSIX.1-2001。线程接口（也称为“pthread”或“POSIX线程”）在POSIX.1-2001中是一个可选特征。POSIX线程的特征测试宏是_POSIX_THREADS，应用程序可以把这个宏用于#ifdef测试，以在编译时确定是否支持线程；也可以把_SC_THREADS常数用于调用sysconf函数，从而在运行时确定是否支持线程。

11.3 线程标识

就像每个进程有一个进程ID一样，每个线程也有一个线程ID。进程ID在整个系统中是唯一的，但线程ID不同，线程ID只在它所属的进程环境中有效。

回忆一下进程ID，它用pid_t数据类型来表示，是一个非负整数。线程ID则用pthread_t数据类型来表示，实现的时候可以用一个结构来代表pthread_t数据类型，所以可移植的操作系统实现不能把它作为整数处理。因此必须使用函数来对两个线程ID进行比较。

356

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```

返回值：若相等则返回非0值，否则返回0

Linux 2.4.22使用无符号长整型数表示pthread_t数据类型。Solaris 9把pthread_t数据类型表示为无符号整数。FreeBSD 5.2.1和Mac OS X 10.3用一个指向pthread结构的指针来表示pthread_t数据类型。

用结构表示pthread_t数据类型的后果是不能用一种可移植的方式打印该数据类型的值。在程序调试过程中打印线程ID有时是非常有用的，而在其他情况下通常不需要打印线程ID，因此有可能出现不可移植的调试代码，当然这也算不上是很大的局限性。

线程可以通过调用pthread_self函数获得自身的线程ID。

```
#include <pthread.h>

pthread_t pthread_self(void);
```

返回值：调用线程的线程ID

当线程需要识别以线程ID作为标识的数据结构时，pthread_self函数可以与pthread_equal函数一起使用。例如，主线程可能把工作任务放在一个队列中，用线程ID来控制每个工作线程处理哪些作业。如图11-1所示，主线程把新的作业放到一个工作队列中，由三个工作线程组成的线程池从队列中移出作业，每个线程并不是任意地处理从队列顶端取出的作业，而是由主线程控制作业的分配，主线程在每个待处理作业的结构中放置处理该作业的线程ID，每个工作线程只能移出标有自己线程ID的作业。

11.4 线程的创建

在传统的UNIX进程模型中，每个进程只有一个控制线程。从概念上讲，这与基于线程的模型中每个进程只包含一个线程是相同的。在POSIX线程（pthread）的情况下，程序开始运行时，它也是以单进程中的单个控制线程启动的，在创建多个控制线程以前，程序的行为与传统的进程并没有什么区别。新增的线程可以通过调用pthread_create函数创建。

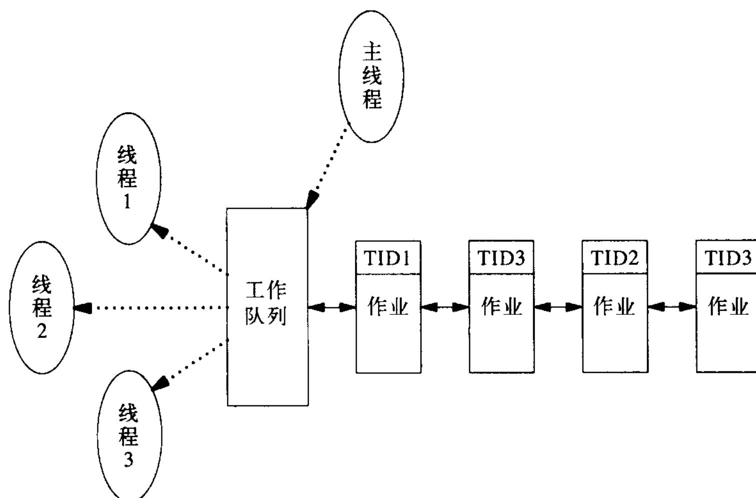


图11-1 工作队列实例

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void), void *restrict arg);
```

返回值：若成功则返回0，否则返回错误编号

357

当pthread_create成功返回时，由*tidp*指向的内存单元被设置为新创建线程的线程ID。*attr*参数用于定制各种不同的线程属性。线程属性将在12.3节中讨论，眼下暂时把它设置为NULL，创建默认属性的线程。

新创建的线程从*start_rtn*函数的地址开始运行，该函数只有一个无类型指针参数*arg*，如果需要向*start_rtn*函数传递的参数不止一个，那么需要把这些参数放到一个结构中，然后把这个结构的地址作为*arg*参数传入。

线程创建时并不能保证哪个线程会先运行：是新创建的线程还是调用线程。新创建的线程可以访问进程的地址空间，并且继承调用线程的浮点环境和信号屏蔽字，但是该线程的未决信号集被清除。

注意pthread函数在调用失败时通常会返回错误码，它们并不像其他的POSIX函数一样设置errno。每个线程都提供errno的副本，这只是为了与使用errno的现有函数兼容。在线程中，从函数中返回错误码更为清晰整洁，不需要依赖那些随着函数执行不断变化的全局状态，因而可以把错误的范围限制在引起出错的函数中。

实例

358

虽然没有可移植的方法打印线程ID，但是可以写一个小的测试程序来完成这个任务，以便更深入地了解线程是如何工作的。程序清单11-1中的程序创建了一个线程并且打印进程ID、新线程的线程ID以及初始线程的线程ID。

程序清单11-1　打印线程ID

```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
      (unsigned int)tid, (unsigned int)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_quit("can't create thread: %s\n", strerror(err));
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

这个实例有两个特别之处，需要处理主线程和新线程之间的竞争。(在本章后面的内容中将学习如何更好地处理这种竞争)。首先是主线程需要休眠，如果主线程不休眠，它就可能退出，这样在新线程有机会运行之前整个进程可能就已经终止了。这种行为特征依赖于操作系统中的线程实现和调度算法。

第二个特别之处在于新线程是通过调用pthread_self函数获取自己的线程ID，而不是从共享内存中读出或者从线程的启动例程中以参数的形式接收到。回忆pthread_create函数，它会通过第一个参数（*tidp*）返回新建线程的线程ID。在本例里，主线程把新线程ID存放在ntid中，但是新建的线程并不能安全地使用它，如果新线程在主线程调用pthread_create返回之前就运行了，那么新线程看到的是未经初始化的ntid的内容，这个内容并不是正确的线程ID。

359

在Solaris上运行程序清单11-1中的程序，得到：

```
$ ./a.out
main thread: pid 7225 tid 1 (0x1)
new thread:  pid 7225 tid 4 (0x4)
```

正如所料，两个线程的进程ID相同，但线程ID不同。在FreeBSD上运行程序清单11-1中的程序，得到：

```
$ ./a.out
main thread: pid 14954 tid 134529024 (0x804c000)
new thread:  pid 14954 tid 134530048 (0x804c400)
```

正如所料，两个线程有相同的进程ID，如果把线程ID看成是十进制整数，线程ID的值看起来很奇怪，但是如果把它们转化成十六进制，就变得有意义多了。就像前面所提及的，FreeBSD使用指向线程数据结构的指针作为它的线程ID。

我们期望Mac OS X与FreeBSD相似，主线程ID与新线程ID所指向的内存单元在相同的地址范围内，但事实上，在Mac OS X中，主线程ID与用pthread_create新创建的线程ID所指向的内存单元并不在相同的地址范围内：

```
$ ./a.out
main thread: pid 779 tid 2684396012 (0xa000a1ec)
new thread:  pid 779 tid 25166336 (0x1800200)
```

相同的程序在Linux上运行得到的结果稍有不同：

```
$ ./a.out
new thread:  pid 6628 tid 1026 (0x402)
main thread: pid 6626 tid 1024 (0x400)
```

Linux线程ID看起来更合理一些，但进程ID并不匹配。这与Linux的线程实现有关，Linux使用clone系统调用来实现pthread_create。clone系统调用创建子进程，这个子进程可以共享父进程一定数量的执行环境（如文件描述符和内存），这个数量是可配置的。

另外还需要注意，主线程的输出基本上出现在新建线程的输出之前，但Linux却不是这样的，所以不能在线程调度上做出任何假设。 □

11.5 线程终止

如果进程中的任一线程调用了exit，_Exit或者_exit，那么整个进程就会终止。与此类似，如果信号的默认动作是终止进程，那么，把该信号发送到线程会终止整个进程（将在12.8节中讨论信号与线程间的交互）。

单个线程可以通过下列三种方式退出，在不终止整个进程的情况下停止它的控制流。 360

(1) 线程只是从启动例程中返回，返回值是线程的退出码。

(2) 线程可以被同一进程中的其他线程取消。

(3) 线程调用pthread_exit。

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

*rval_ptr*是一个无类型指针，与传给启动例程的单个参数类似。进程中的其他线程可以通过调用pthread_join函数访问到这个指针。

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);

                        返回值：若成功则返回0，否则返回错误编号
```

调用线程将一直阻塞，直到指定的线程调用pthread_exit、从启动例程中返回或者被取消。

如果线程只是从它的启动例程返回，*rval_ptr*将包含返回码。如果线程被取消，由*rval_ptr*指定的内存单元就置为PTHREAD_CANCELED。

可以通过调用pthread_join自动把线程置于分离状态（马上就会讨论到），这样资源就可以恢复。如果线程已经处于分离状态，pthread_join调用就会失败，返回EINVAL。

如果对线程的返回值并不感兴趣，可以把*rval_ptr*置为NULL。在这种情况下，调用pthread_join函数将等待指定的线程终止，但并不获取线程的终止状态。

程序清单11-2说明了如何获取已终止的线程的退出码。

程序清单11-2 获得线程退出状态

```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

运行程序清单11-2中的程序，得到的结果是：

```
$ ./a.out
```

```
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

可以看出，当一个线程通过调用pthread_exit退出或者简单地从启动例程中返回时，进程中的其他线程可以通过调用pthread_join函数获得该线程的退出状态。 □

pthread_create和pthread_exit函数的无类型指针参数能传递的数值可以不止一个，该指针可以传递包含更复杂信息的结构的地址，但是注意这个结构所使用的内存在调用者完成调用以后必须仍然是有效的，否则就会出现无效或非法内存访问。例如，在调用线程的栈上分配了该结构，那么其他的线程在使用这个结构时内存内容可能已经改变了。又如，线程在自己的栈上分配了一个结构然后把指向这个结构的指针传给pthread_exit，那么当调用pthread_join的线程试图使用该结构时，这个栈有可能已经被撤销，这块内存也已另作他用。 [362]

程序清单11-3中的程序给出了用自动变量（分配在栈上）作为pthread_exit的参数时出现的问题。

程序清单11-3 pthread_exit参数的不正确使用

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf(s);
    printf("  structure at 0x%x\n", (unsigned)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo  foo = {1, 2, 3, 4};

    printfoo("thread 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2: ID is %d\n", pthread_self());
    pthread_exit((void *)0);
}

int
main(void)
{
```

```
    int          err;
    pthread_t    tid1, tid2;
    struct foo   *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    sleep(1);
    printf("parent starting second thread\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    sleep(1);
    printfoo("parent:\n", fp);
    exit(0);
}
```

363

在Linux上运行此程序，得到

```
$ ./a.out
thread 1:
  structure at 0x409a2abc
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 32770
parent:
  structure at 0x409a2abc
  foo.a = 0
  foo.b = 32770
  foo.c = 1075430560
  foo.d = 1073937284
```

当然，运行结果根据内存结构、编译器以及线程库的实现会有所不同。在FreeBSD上的结果类似于

```
$ ./a.out
thread 1:
  structure at 0xbfafefc0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 134534144
parent:
  structure at 0xbfafefc0
  foo.a = 0
  foo.b = 134534144
  foo.c = 3
  foo.d = 671642590
```

可以看出，当主线程访问这个结构时，结构的内容（在线程*tid1*的栈上分配）已经改变。注意第二个线程（*tid2*）的栈是如何覆盖第一个线程的栈的。为了解决这个问题，可以使用全局结构，或者用malloc函数分配结构。□

364

线程可以通过调用pthread_cancel函数来请求取消同一进程中的其他线程。

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

返回值：若成功则返回0，否则返回错误编号

在默认的情况下，pthread_cancel函数会使得由*tid*标识的线程的行为表现为如同调用了参数为PTHREAD_CANCELED的pthread_exit函数，但是，线程可以选择忽略取消方式或是控制取消方式。相关内容将在12.7节中详细讨论。注意pthread_cancel并不等待线程终止，它仅仅提出请求。

线程可以安排它退出时需要调用的函数，这与进程可以用atexit函数（见7.3节）安排进程退出时需要调用的函数是类似的。这样的函数称为*线程清理处理程序*(thread cleanup handler)。线程可以建立多个清理处理程序。处理程序记录在栈中，也就是说它们的执行顺序与它们注册时的顺序相反。

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

当线程执行以下动作时调用清理函数，调用参数为*arg*，清理函数*rtn*的调用顺序是由pthread_cleanup_push函数来安排的。

- 调用pthread_exit时。
- 响应取消请求时。
- 用非零*execute*参数调用pthread_cleanup_pop时。

如果*execute*参数置为0，清理函数将不被调用。无论哪种情况，pthread_cleanup_pop都将删除上次pthread_cleanup_push调用建立的清理处理程序。

这些函数有一个限制，由于它们可以实现为宏，所以必须在与线程相同的作用域中以匹配对的形式使用，pthread_cleanup_push的宏定义可包含字符{，在这种情况下对应的匹配字符}就要在pthread_cleanup_pop定义中出现。

程序清单11-4显示了如何使用线程清理处理程序。虽然例子有人为编造之嫌，但说清楚了其中涉及的清理机制。注意，虽然并未打算要传一个非零参数给线程启动例程，还是需要把pthread_cleanup_pop调用和pthread_cleanup_push调用匹配起来，否则，程序编译可能通不过。 365

程序清单11-4 线程清理处理程序

```
#include "apue.h"
#include <pthread.h>

void
cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
```

```
}

void *
thr_fn1(void *arg)
{
    printf("thread 1 start\n");
    pthread_cleanup_push(cleanup, "thread 1 first handler");
    pthread_cleanup_push(cleanup, "thread 1 second handler");
    printf("thread 1 push complete\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 start\n");
    pthread_cleanup_push(cleanup, "thread 2 first handler");
    pthread_cleanup_push(cleanup, "thread 2 second handler");
    printf("thread 2 push complete\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

366

运行程序清单11-4中的程序会得到：

```
$ ./a.out
thread 1 start
thread 1 push complete
thread 2 start
thread 2 push complete
```

```
cleanup: thread 2 second handler
cleanup: thread 2 first handler
thread 1 exit code 1
thread 2 exit code 2
```

从输出结果可以看出，两个线程都正确地启动和退出了，但是只调用了第二个线程的清理处理程序，所以如果线程是通过从它的启动例程中返回而终止的话，那么它的清理处理程序就不会被调用，还要注意清理处理程序是按照与它们安装时相反的顺序被调用的。 □

现在可以开始看出线程函数和进程函数之间的相似之处。表11-1总结了这些相似的函数。

表11-1 进程原语和线程原语的比较

进程原语	线程原语	描 述
fork	pthread_create	创建新的控制流
exit	pthread_exit	从现有的控制流中退出
waitpid	pthread_join	从控制流中得到退出状态
atexit	pthread_cancel_push	注册在退出控制流时调用的函数
getpid	pthread_self	获取控制流的ID
abort	pthread_cancel	请求控制流的非正常退出

在默认情况下，线程的终止状态会保存到对该线程调用pthread_join，如果线程已经处于分离状态，线程的底层存储资源可以在线程终止时立即被收回。当线程被分离时，并不能用pthread_join函数等待它的终止状态。对分离状态的线程进行pthread_join的调用会产生失败，返回EINVAL。pthread_detach调用可以用于使线程进入分离状态。 367

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

返回值：若成功则返回0，否则返回错误编号

在下一章中，将学习通过对传给pthread_create函数的线程属性进行修改，创建一个已处于分离状态的线程。

11.6 线程同步

当多个控制线程共享相同的内存时，需要确保每个线程看到一致的数据视图。如果每个线程使用的变量都是其他线程不会读取或修改的，那么就不存在一致性问题。同样地，如果变量是只读的，多个线程同时读取该变量也不会有一致性问题。但是，当某个线程可以修改变量，而其他线程也可以读取或者修改这个变量的时候，就需要对这些线程进行同步，以确保它们在访问变量的存储内容时不会访问到无效的数值。

当一个线程修改变量时，其他线程在读取这个变量的值时就可能会看到不一致的数据。 在变量修改时间多于一个存储器访问周期的处理器结构中，当存储器读与存储器写这两个周期交叉时，这种潜在的不一致性就会出现。当然，这种行为是与处理器结构相关的，但是可移植的程序并不能对使用何种处理器结构做出假设。

图11-2描述了两个线程读写相同变量的假设例子。在这个例子中，线程A读取变量然后给这个变量赋予一个新的值，但写操作需要两个存储器周期。当线程B在这两个存储器写周期中间读取这个相同的变量时，它就会得到不一致的值。

为了解决这个问题，线程不得不使用锁，在同一时间只允许一个线程访问该变量。图11-3描述了这种同步。如果线程B希望读取变量，它首先要获取锁；同样地，当线程A更新变量时，也需要获取这把同样的锁。因而线程B在线程A释放锁以前不能读取变量。

368

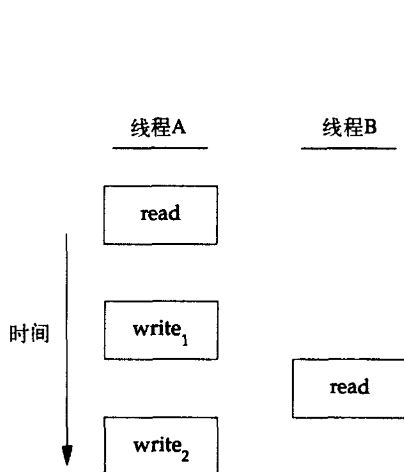


图11-2　两个线程的交叉存储器周期

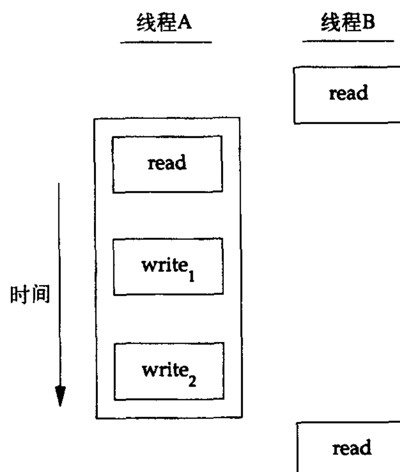


图11-3　两个线程同步内存访问

当两个或多个线程试图在同一时间修改同一变量时，也需要进行同步。考虑变量递增操作的情况（图11-4），增量操作通常可分为三步：

(1) 从内存单元读入寄存器。

(2) 在寄存器中进行变量值的增加。

(3) 把新的值写回内存单元。

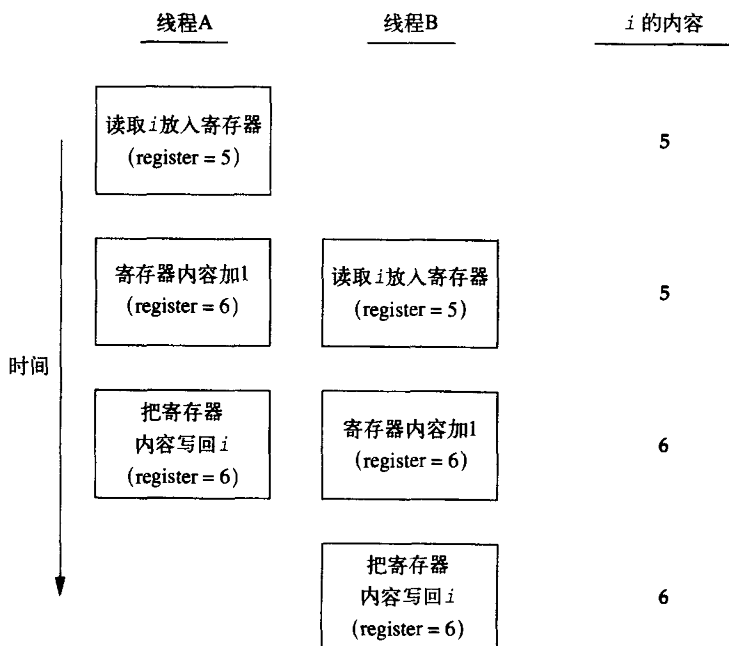


图11-4　两个非同步的线程对同一个变量做增量操作

如果两个线程试图在几乎同一时间对同一个变量做增量操作而不进行同步的话，结果就可能出现不一致。变量可能比原来增加了1，也有可能比原来增加了2，具体是1还是2取决于第二个线程开始操作时获取的数值。如果第二个线程执行第一步要比第一个线程执行第三步早，第二个线程读到的初始值就与第一个线程一样，它为变量加1，然后再写回去，事实上没有实际的效果，总的来说变量只增加了1。

如果修改操作是原子操作，那么就不存在竞争。在前面的例子中，如果增加1只需要一个存储器周期，那么就没有竞争存在。如果数据总是以*顺序一致*的方式出现，就不需要额外的同步。当多个线程并不能观察到数据的不一致时，那么操作就是顺序一致的。在现代计算机系统中，存储器访问需要多个总线周期，多处理器的总线周期通常在多个处理器上是交叉的，所以无法保证数据是顺序一致的。 369

在顺序一致的环境中，可以把数据修改操作解释为运行线程的顺序操作步骤。可以把这样的情形描述为“线程A对变量增加了1，然后线程B对变量增加了1，所以变量的值比原来的大2”，或者描述为“线程B对变量增加了1，然后线程A对变量增加了1，所以变量的值比原来的大2”。这两个线程的任何操作顺序都不可能让变量出现除了上述值以外的其他数值。

除了计算机体系结构的因素以外，程序使用变量的方式也会引起竞争，也会导致不一致的情况发生。例如，可能会对某个变量加1，然后基于这个数值做出某种决定。增量操作这一步和做出决定这一步两者的组合并非原子操作，因而给不一致情况的出现提供了可能。

1. 互斥量

可以通过使用pthread的互斥接口保护数据，确保同一时间只有一个线程访问数据。*互斥量*（mutex）从本质上说是一把锁，在访问共享资源前对互斥量进行加锁，在访问完成后释放互斥量上的锁。对互斥量进行加锁以后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。如果释放互斥锁时有多个线程阻塞，所有在该互斥锁上的阻塞线程都会变成可运行状态，第一个变为运行状态的线程可以对互斥量加锁，其他线程将会看到互斥锁依然被锁住，只能回去再次等待它重新变为可用。在这种方式下，每次只有一个线程可以向前执行。 370

在设计时需要规定所有的线程必须遵守相同的数据访问规则，只有这样，互斥机制才能正常工作。操作系统并不会做数据访问的串行化。如果允许其中的某个线程在没有得到锁的情况下也可以访问共享资源，那么即使其他的线程在使用共享资源前都获取了锁，也还是会出现数据不一致的问题。

互斥变量用`pthread_mutex_t`数据类型来表示，在使用互斥变量以前，必须首先对它进行初始化，可以把它置为常量`PTHREAD_MUTEX_INITIALIZER`（只对静态分配的互斥量），也可以通过调用`pthread_mutex_init`函数进行初始化。如果动态地分配互斥量（例如通过调用`malloc`函数），那么在释放内存前需要调用`pthread_mutex_destroy`。

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

返回值：若成功则返回0，否则返回错误编号

要用默认的属性初始化互斥量，只需把*attr*设置为`NULL`。非默认的互斥量属性将在12.4节中

讨论。

对互斥量进行加锁，需要调用pthread_mutex_lock，如果互斥量已经上锁，调用线程将阻塞直到互斥量被解锁。对互斥量解锁，需要调用pthread_mutex_unlock。

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

返回值：若成功则返回0，否则返回错误编号

如果线程不希望被阻塞，它可以使用pthread_mutex_trylock尝试对互斥量进行加锁。如果调用pthread_mutex_trylock时互斥量处于未锁住状态，那么pthread_mutex_trylock将锁住互斥量，不会出现阻塞并返回0，否则pthread_mutex_trylock就会失败，不能锁住互斥量，而返回EBUSY。

程序清单11-5描述了用于保护某个数据结构的互斥量。当多个线程需要访问动态分配的对象时，可以在对象中嵌入引用计数，确保在所有使用该对象的线程完成数据访问之前，该对象内存空间不会被释放。

371

程序清单11-5 使用互斥量保护数据结构

```
#include <stdlib.h>
#include <pthread.h>

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
```

```
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

在对引用计数加1、减1以及检查引用计数是否为0这些操作之前需要锁住互斥量。在foo_alloc函数将引用计数初始化为1时没必要加锁，因为在这个操作之前分配线程是唯一引用该对象的线程。但是在这之后如果要将该对象放到一个列表中，那么它就有可能被别的线程发现，因此需要首先对它加锁。

372

在使用该对象前，线程需要对这个对象的引用计数加1，当对象使用完毕时，需要对引用计数减1。当最后一个引用被释放时，对象所占的内存空间就被释放。 □

2. 避免死锁

如果线程试图对同一个互斥量加锁两次，那么它自身就会陷入死锁状态，使用互斥量时，还有其他更不明显的方式也能产生死锁。例如，程序中使用多个互斥量时，如果允许一个线程一直占有第一个互斥量，并且在试图锁住第二个互斥量时处于阻塞状态，但是拥有第二个互斥量的线程也在试图锁住第一个互斥量，这时就会发生死锁。因为两个线程都在相互请求另一个线程拥有的资源，所以这两个线程都无法向前运行，于是就产生死锁。

可以通过小心地控制互斥量加锁的顺序来避免死锁的发生。例如，假设需要对两个互斥量A和B同时加锁，如果所有线程总是在对互斥量B加锁之前锁住互斥量A，那么使用这两个互斥量不会产生死锁（当然在其他的资源上仍可能出现死锁）；类似地，如果所有的线程总是在锁住互斥量A之前锁住互斥量B，那么也不会发生死锁。只有在一个线程试图以与另一个线程相反的顺序锁住互斥量时，才可能出现死锁。

有时候应用程序的结构使得对互斥量加锁进行排序是很困难的，如果涉及了太多的锁和数据结构，可用的函数并不能把它转换成简单的层次，那么就需要采用另外的方法。可以先释放占有的锁，然后过一段时间再试。这种情况可以使用pthread_mutex_trylock接口避免死锁。如果已经占有某些锁而且pthread_mutex_trylock接口返回成功，那么就可以前进；但是，如果不能获取锁，可以先释放已经占有的锁，做好清理工作，然后过一段时间重新尝试。

在这个例子中，我们修改了程序清单11-5，用以描述两个互斥量的使用方法（见程序清单11-6）。当同时需要两个互斥量时，总是让它们以相同的顺序加锁，以避免死锁。第二个互斥量维护着一个用于跟踪foo数据结构的散列列表。这样hashlock互斥量保护foo数据结构中的fh散列表和f_next 散列链字段。foo结构中的f_lock互斥量保护对foo结构中的其他字段的访问。

程序清单11-6 使用两个互斥量

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)
struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    struct foo     *f_next; /* protected by hashlock */
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

struct foo *
foo_find(int id) /* find an existing object */
{
    struct foo  *fp;
    int         idx;

    idx = HASH(fp);
    pthread_mutex_lock(&hashlock);
    for (fp = fh[idx]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
```

373 374

```
            foo_hold(fp);
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* need to recheck the condition */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* remove from list */
        idx = HASH(fp);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count--;
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

375

比较程序清单11-6和程序清单11-5，可以看出分配函数现在锁住散列列表锁，把新的结构添加到散列存储桶中，在对散列列表的锁解锁之前，先锁住新结构中的互斥量。因为新的结构是放在全局列表中的，其他线程可以找到它，所以在完成初始化之前，需要阻塞其他试图访问新结构的线程。

foo_find函数锁住散列列表锁然后搜索被请求的结构。如果找到了，就增加其引用计数并返回指向该结构的指针。注意加锁的顺序是先在foo_find函数中锁定散列列表锁，然后再在foo_hold函数中锁定foo结构中的f_lock互斥量。

现在有了两个锁以后，foo_rele函数变得更加复杂。如果这是最后一个引用，因为将需要从散列列表中删除这个结构，就要先对这个结构互斥量进行解锁，才可以获取散列列表锁。然后重新获取结构互斥量。从上一次获得结构互斥量以来可能处于被阻塞状态，所以需要重新

检查条件，判断是否还需要释放这个结构。如果其他线程在我们为满足锁顺序而阻塞时发现了这个结构并对其引用计数加1，那么只需要简单地对引用计数减1，对所有的东西解锁然后返回。

如此加、解锁太复杂，所以需要重新审视原来的设计。也可以使用散列列表锁来保护结构引用计数，使事情大大简化，结构互斥量可以用于保护foo结构中的其他任何东西。程序清单11-7反应了这种变化。

程序清单11-7　简化的加、解锁

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count; /* protected by hashlock */
    pthread_mutex_t f_lock;
    struct foo     *f_next; /* protected by hashlock */
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&hashlock);
    fp->f_count++;
    pthread_mutex_unlock(&hashlock);
}

struct foo *
foo_find(int id) /* find a existing object */
{
```

376

```
    struct foo  *fp;
    int         idx;

    idx = HASH(fp);
    pthread_mutex_lock(&hashlock);
    for (fp = fh[idx]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            fp->f_count++;
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&hashlock);
    if (--fp->f_count == 0) { /* last reference, remove from list */
        idx = HASH(fp);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&hashlock);
    }
}
```

377

注意，与程序清单11-6中的程序相比，程序清单11-7中的程序简单得多。两种用途使用相同的锁时，围绕散列列表和引用计数的锁的排序问题就随之不见了。多线程的软件设计经常要考虑这类折中处理方案。如果锁的粒度太粗，就会出现很多线程阻塞等待相同的锁，源自并发性的改善微乎其微。如果锁的粒度太细，那么过多的锁开销会使系统性能受到影响，而且代码变得相当复杂。作为一个程序员，需要在满足锁需求的情况下，在代码复杂性和优化性能之间找好平衡点。□

3. 读写锁

读写锁与互斥量类似，不过读写锁允许更高的并行性。互斥量要么是锁住状态要么是不加锁状态，而且一次只有一个线程可以对其加锁。读写锁可以有三种状态：读模式下加锁状态，写模式下加锁状态，不加锁状态。一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁。

当读写锁是写加锁状态时，在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞。当读写锁在读加锁状态时，所有试图以读模式对它进行加锁的线程都可以得到访问权，但是如果线程希望以写模式对此锁进行加锁，它必须阻塞直到所有的线程释放读锁。虽然读写锁的实现各不相同，但当读写锁处于读模式锁住状态时，如果有另外的线程试图以写模式加锁，

读写锁通常会阻塞随后的读模式锁请求。这样可以避免读模式锁长期占用，而等待的写模式锁请求一直得不到满足。

读写锁非常适合于对数据结构读的次数远大于写的情况。当读写锁在写模式下时，它所保护的数据结构就可以被安全地修改，因为当前只有一个线程可以在写模式下拥有这个锁。当读写锁在读模式下时，只要线程获取了读模式下的读写锁，该锁所保护的数据结构可以被多个获得读模式锁的线程读取。

378

读写锁也叫做共享－独占锁，当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。

与互斥量一样，读写锁在使用之前必须初始化，在释放它们底层的内存前必须销毁。

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

读写锁通过调用pthread_rwlock_init进行初始化。如果希望读写锁有默认的属性，可以传一个空指针给*attr*，读写锁的属性将在12.4节中讨论。

在释放读写锁占用的内存之前，需要调用pthread_rwlock_destroy做清理工作。如果pthread_rwlock_init为读写锁分配了资源，pthread_rwlock_destroy将释放这些资源。如果在调用pthread_rwlock_destroy之前就释放了读写锁占用的内存空间，那么分配给这个锁的资源就丢失了。

要在读模式下锁定读写锁，需要调用pthread_rwlock_rdlock；要在写模式下锁定读写锁，需要调用pthread_rwlock_wrlock。不管以何种方式锁住读写锁，都可以调用pthread_rwlock_unlock进行解锁。

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

所有的返回值都是：若成功则返回0，否则返回错误编号

在实现读写锁的时候可能会对共享模式下可获取的锁的数量进行限制，所以需要检查pthread_rwlock_rdlock的返回值。即使pthread_rwlock_wrlock和 pthread_rwlock_unlock有错误的返回值，如果锁设计合理的话，也不需要检查其返回值。错误返回值的定义只是针对不正确地使用读写锁的情况，例如未经初始化的锁，或者试图获取已拥有的锁从而可能产生死锁这样的错误返回等。

Single UNIX Specification同样定义了有条件的读写锁原语的版本。

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

379

可以获取锁时，函数返回0；否则，返回错误EBUSY。这些函数可以用于前面讨论的遵守某种锁层次但还不能完全避免死锁的情况。

程序清单11-8中的程序解释了读写锁的使用。作业请求队列由单个读写锁保护。这个例子给出了图11-1的可能的一种实现，以此实现多个工作线程获取由单个主线程分配给它们的作业。

程序清单11-8 使用读写锁

```
#include <stdlib.h>
#include <pthread.h>

struct job {
    struct job *j_next;
    struct job *j_prev;
    pthread_t   j_id;   /* tells which thread handles this job */
    /* ... more stuff here ... */
};

struct queue {
    struct job      *q_head;
    struct job      *q_tail;
    pthread_rwlock_t q_lock;
};

/*
 * Initialize a queue.
 */
int
queue_init(struct queue *qp)
{
    int err;

    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);

    /* ... continue initialization ... */

    return(0);
}

/*
 * Insert a job at the head of the queue.
 */
void
job_insert(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp;    /* list was empty */
    qp->q_head = jp;
```

380

```
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Append a job on the tail of the queue.
 */
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp;    /* list was empty */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Remove the given job from a queue.
 */
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp)
            qp->q_tail = NULL;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        if (qp->q_head == jp)
            qp->q_head = NULL;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}
```

381

```
/*
 * Find a job for the given thread ID.
 */
struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);

    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
        if (pthread_equal(jp->j_id, id))
            break;

    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

在这个例子中，不管什么时候需要增加一个作业到队列中或者从队列中删除作业，都用写模式锁住队列的读写锁。不管何时搜索队列，首先需要获取读模式下的锁，允许所有的工作线程并发地搜索队列。在这种情况下，只有线程搜索队列的频率远远高于增加或删除作业时，使用读写锁才可能改善性能。

工作线程只能从队列中读取与它们的线程ID匹配的作业。既然作业结构同一时间只能由一个线程使用，所以不需要额外加锁。 □

4. 条件变量

条件变量是线程可用的另一种同步机制。条件变量给多个线程提供了一个会合的场所。条件变量与互斥量一起使用时，允许线程以无竞争的方式等待特定的条件发生。

条件本身是由互斥量保护的。线程在改变条件状态前必须首先锁住互斥量，其他线程在获得互斥量之前不会察觉到这种改变，因为必须锁定互斥量以后才能计算条件。

条件变量使用之前必须首先进行初始化，pthread_cond_t数据类型代表的条件变量可以用两种方式进行初始化，可以把常量PTHREAD_COND_INITIALIZER赋给静态分配的条件变量，但是如果条件变量是动态分配的，可以使用pthread_cond_init函数进行初始化。

在释放底层的内存空间之前，可以使用pthread_mutex_destroy函数对条件变量进行去除初始化（deinitialize）。

382

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond);

                    两者的返回值都是：若成功则返回0，否则返回错误编号
```

除非需要创建一个非默认属性的条件变量，否则pthread_cond_init函数的*attr*参数可以设置为NULL，条件变量属性将在12.4节中讨论。

使用pthread_cond_wait等待条件变为真，如果在给定的时间内条件不能满足，那么会生成一个代表出错码的返回变量。

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict timeout);

                    两者的返回值都是：若成功则返回0，否则返回错误编号
```

传递给pthread_cond_wait的互斥量对条件进行保护，调用者把锁住的互斥量传给函数。函数把调用线程放到等待条件的线程列表上，然后对互斥量解锁，这两个操作是原子操作。这样就关闭了条件检查和线程进入休眠状态等待条件改变这两个操作之间的时间通道，这样线程就不会错过条件的任何变化。pthread_cond_wait返回时，互斥量再次被锁住。

pthread_cond_timedwait函数的工作方式与pthread_cond_wait函数相似，只是多了一个*timeout*。*timeout*值指定了等待的时间，它是通过timespec结构指定。时间值用秒数

或者分秒数来表示，分秒数的单位是纳秒。

```
struct timespec {
        time_t tv_sec;    /* seconds */
        long   tv_nsec;   /* nanoseconds */
};
```

使用这个结构时，需要指定愿意等待多长时间，时间值是一个绝对数而不是相对数。例如，如果能等待3分钟，就需要把当前时间加上3分钟再转换到timespec结构，而不是把3分钟转换成timespec结构。

可以使用gettimeofday（见6.10节）获取用timeval结构表示的当前时间，然后把这个时间转换成timespec结构。要得到timeout值的绝对时间，可以使用下面的函数：

383

```
void
maketimeout(struct timespec *tsp, long minutes)
{
    struct timeval now;

    /* get the current time */
    gettimeofday(&now);
    tsp->tv_sec = now.tv_sec;
    tsp->tv_nsec = now.tv_usec * 1000; /* usec to nsec */
    /* add the offset to get timeout value */
    tsp->tv_sec += minutes * 60;
}
```

如果时间值到了但是条件还是没有出现，pthread_cond_timedwait将重新获取互斥量然后返回错误ETIMEDOUT。从pthread_cond_wait或者pthread_cond_timedwait调用成功返回时，线程需要重新计算条件，因为其他的线程可能已经在运行并改变了条件。

有两个函数可以用于通知线程条件已经满足。pthread_cond_signal函数将唤醒等待该条件的某个线程，而pthread_cond_broadcast函数将唤醒等待该条件的所有线程。

> POSIX规范为了简化实现，允许pthread_cond_signal在实现的时候可以唤醒不止一个线程。

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

调用pthread_cond_signal或者pthread_cond_broadcast，也称为向线程或条件发送信号。必须注意一定要在改变条件状态以后再给线程发送信号。

实　例

程序清单11-9给出了如何结合使用条件变量和互斥量对线程进行同步。

程序清单11-9　使用条件变量

```
#include <pthread.h>

struct msg {
```

```
    struct msg *m_next;
    /* ... more stuff here ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* now process the message mp */
    }
}

void
enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

384

条件是工作队列的状态。用互斥量保护条件，在while循环中判断条件。把消息放到工作队列时，需要占有互斥量，但向等待线程发送信号时并不需要占有互斥量。只要线程可以在调用cond_signal之前把消息从队列中拖出，就可以在释放互斥量以后再完成这部分工作。因为是在while循环中检查条件，所以不会存在问题：线程醒来，发现队列仍为空，然后返回继续等待。如果代码不能容忍这种竞争，就需要在向线程发送信号的时候占有互斥量。□

11.7 小结

在本章中，介绍了线程的概念，讨论了现有的创建线程和销毁线程的POSIX.1原语；此外还介绍了线程同步问题，讨论了三种基本的同步机制：互斥、读写锁以及条件变量，了解了如何使用它们来保护共享资源。

385

习题

11.1 修改程序清单11-3中的例子，从而正确地在线程之间传递结构。

11.2 在程序清单11-8的例子中，需要另外添加什么同步（如果需要的话）才可以使得主线程改变与未决作业关联的线程ID？这会对job_remove函数产生什么影响？

11.3 把程序清单11-9中的技术运用到工作线程实例（图11-1和程序清单11-8）中，以实现工作线程函数。不要忘记更新queue_init函数对条件变量进行初始化，修改job_insert

和job_append函数通知工作线程。会出现什么样的困难?

11.4 下面哪个步骤序列是正确的?

(1) 对互斥量加锁 (pthread_mutex_lock)。

(2) 改变互斥量保护的条件。

(3) 向等待条件的线程发送信号 (pthread_cond_broadcast)。

(4) 对互斥量解锁 (pthread_mutex_unlock)。

或者

(1) 对互斥量加锁 (pthread_mutex_lock)。

(2) 改变互斥量保护的条件。

(3) 对互斥量解锁 (pthread_mutex_unlock)。

386

(4) 向等待条件的线程发送信号 (pthread_cond_broadcast)。

第12章

线程控制

12.1 引言

在第11章中，学习了线程和线程同步的基础知识。在本章中，将学习控制线程行为方面的详细内容，在前面的章节中对线程属性和同步原语属性都取其默认行为，忽略了这些属性的具体介绍。

接下来将介绍同一进程中的多个线程之间如何保持数据的私有性，最后讨论基于进程的系统调用如何与线程进行交互。

12.2 线程限制

在2.5.4节中讨论了sysconf函数，Single UNIX Specification定义了与线程操作有关的一些限制，表2-10并没有列出这些限制。与其他的系统限制一样，这些线程限制也可以通过sysconf函数进行查询。表12-1总结了这些限制。

表12-1 线程限制和sysconf的*name*参数

限制名称	描　述	*name*参数
PTHREAD_DESTRUCTOR_ITERATIONS	线程退出时操作系统实现试图销毁线程私有数据的最大次数（见12.6节）	_SC_THREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX	进程可以创建的键的最大数目（见12.6节）	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	一个线程的栈可用的最小字节数（见12.3节）	_SC_THREAD_STACK_MIN
PTHREAD_THREADS_MAX	进程可以创建的最大线程数（见12.3节）	_SC_THREAD_THREADS_MAX

与sysconf报告的其他限制一样，这些限制的使用是为了增强应用程序在不同的操作系统实现之间的可移植性。例如，如果应用程序需要为它管理的每个文件创建四个线程，但是系统却并不允许创建所有这些线程，这时可能就必须限制当前可并发管理的文件数。

387

表12-2给出了本书描述的四种操作系统实现中线程限制的值。当某些操作系统实现没有定义相应的sysconf符号（以_SC_开头）时，图中列出的值就是“未定义符号”；如果操作系统实现的限制是不确定的，列出的值就是“没有确定的限制”，但这并不意味着值是无限制的；“不支持”表明操作系统实现定义了相应的sysconf限制符号，但是sysconf函数无法识别这个符号。

> 注意，虽然某些操作系统实现可能没有提供访问这些限制的方法，但这并不意味着这些限制不存在，它只是表明操作系统实现没有提供使用sysconf访问这些值的方法。

表12-2 线程配置限制的例子

限　制	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
PTHREAD_DESTRUCTOR_ITERATIONS	未定义符号	不支持	未定义符号	没有确定的限制
PTHREAD_KEYS_MAX	未定义符号	不支持	未定义符号	没有确定的限制
PTHREAD_STACK_MIN	未定义符号	不支持	未定义符号	4 096
PTHREAD_THREADS_MAX	未定义符号	不支持	未定义符号	没有确定的限制

12.3 线程属性

在第11章所有调用pthread_create函数的例子中，传入的参数都是空指针，而不是指向pthread_attr_t结构的指针。可以使用pthread_attr_t结构修改线程默认属性，并把这些属性与创建的线程联系起来。可以使用pthread_attr_init函数初始化pthread_attr_t结构。调用pthread_attr_init以后，pthread_attr_t结构所包含的内容就是操作系统实现支持的线程所有属性的默认值。如果要修改其中个别属性的值，需要调用其他的函数，这方面的细节将在本节的后续内容中讨论。

388

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);

                    两者的返回值都是：若成功则返回0，否则返回错误编号
```

如果要去除对pthread_attr_t结构的初始化，可以调用pthread_attr_destroy函数。如果pthread_attr_init实现时为属性对象分配了动态内存空间，pthread_attr_destroy将会释放该内存空间。除此之外，pthread_attr_destroy还会用无效的值初始化属性对象，因此如果该属性对象被误用，将会导致pthread_create函数返回错误。

pthread_attr_t结构对应用程序是不透明的，也就是说应用程序并不需要了解有关属性对象内部结构的任何细节，因而可以增强应用程序的可移植性。POSIX.1沿用了这种模型，并且为查询和设置每种属性定义了独立的函数。

表12-3总结了POSIX.1定义的线程属性。虽然POSIX.1还为实时线程定义了额外的属性，但这里并不打算讨论这些属性。表12-3同时给出了操作系统平台对每种线程属性的支持情况。如果某些属性是通过过时的接口进行访问的，则在表中用**ob**表示。

表12-3 POSIX.1 线程属性

名　称	描　述	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
detachstate	线程的分离状态属性	•	•	•	•
guardsize	线程栈末尾的警戒缓冲区大小（字节数）		•	•	•
stackaddr	线程栈的最低地址	ob	•	•	ob
stacksize	线程栈的大小（字节数）	•	•	•	•

11.5节介绍了分离线程的概念。如果对现有的某个线程的终止状态不感兴趣的话，可以使用pthread_detach函数让操作系统在线程退出时收回它所占用的资源。

如果在创建线程时就知道不需要了解线程的终止状态，则可以修改pthread_attr_t结构中的*detachstate*线程属性，让线程以分离状态启动。可以使用pthread_attr_setdetachstate函数把线程属性*detachstate*设置为下面的两个合法值之一：设置为PTHREAD_CREATE_DETACHED，以分离状态启动线程；或者设置为PTHREAD_CREATE_JOINABLE，正常启动线程，应用程序可以获取线程的终止状态。

389

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

可以调用pthread_attr_getdetachstate函数获取当前的*detachstate*线程属性，第二个参数所指向的整数也许被设置为PTHREAD_CREATE_DETACHED，也可能设置为PTHREAD_CREATE_JOINABLE，具体要取决于给定pthread_attr_t结构中的属性值。

程序清单12-1给出了一个以分离状态创建线程的函数。

程序清单12-1 以分离状态创建的线程

```
#include "apue.h"
#include <pthread.h>

int
makethread(void *(*fn)(void *), void *arg)
{
    int             err;
    pthread_t       tid;
    pthread_attr_t  attr;

    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

注意，这里忽略了pthread_attr_destroy函数调用的返回值。在这种情况下，由于对线程属性进行了合理的初始化，pthread_attr_destroy一般不会失败。但是如果pthread_attr_destroy确实出现了失败的情况，清理工作就会变得很困难：必须销毁刚刚创建的线程，而这个线程可能已经运行，并且与pthread_attr_destroy函数可能是异步执行的。忽略pthread_attr_destroy的错误返回可能出现的最坏情况是：如果pthread_attr_init分配了内存空间，这些内存空间会被泄漏。另一方面，如果pthread_attr_init成功地对线程属性进行了初始化，但pthread_attr_destroy在做清理工作时却出现

了失败，就没有任何补救策略，因为线程属性结构对应用来说是不透明的，可以对线程属性结构进行清理的唯一接口是pthread_attr_destroy，但它失败了。 □

390

对于遵循POSIX标准的操作系统来说，并不一定要支持线程栈属性，但是对遵循XSI的系统，支持线程栈属性就是必须的。可以在编译阶段使用_POSIX_THREAD_ATTR_STACKADDR和_POSIX_THREAD_ATTR_STACKSIZE符号来检查系统是否支持线程栈属性，如果系统定义了这些符号，就说明它支持相应的线程栈属性。也可以通过在运行阶段把_SC_THREAD_ATTR_STACKADDR和_SC_THREAD_ATTR_STACKSIZE参数传给sysconf函数，检查系统对线程栈属性的支持情况。

POSIX.1定义了线程栈属性的一些操作接口。虽然很多pthread实现中仍然提供两个早些时候的函数pthread_attr_getstackaddr和pthread_attr_setstackaddr，但在Single UNIX Specification第3版中这两个函数已被标记为过时，线程栈属性的查询和修改一般是通过较新的函数pthread_attr_getstack和pthread_attr_setstack来进行。这些新的函数消除了老接口定义中存在的二义性。

```
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(const pthread_attr_t *attr,
                          void *stackaddr, size_t *stacksize);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

这两个函数可以用于管理*stackaddr*线程属性，也可以用于管理*stacksize*线程属性。

对进程来说，虚拟地址空间的大小是固定的，进程中只有一个栈，所以它的大小通常不是问题。但对线程来说，同样大小的虚拟地址空间必须被所有的线程栈共享。如果应用程序使用了太多的线程，致使线程栈的累计大小超过了可用的虚拟地址空间，这时就需要减少线程默认的栈大小。另一方面，如果线程调用的函数分配了大量的自动变量或者调用的函数涉及很深的栈帧（stack frame），那么这时需要的栈大小可能要比默认的大。

如果用完了线程栈的虚拟地址空间，可以使用malloc或者mmap（见14.9节）来为其他栈分配空间，并用pthread_attr_setstack函数来改变新建线程的栈位置。线程栈所占内存范围中可寻址的最低地址可以由*stackaddr*参数指定，该地址与处理器结构相应的边界对齐。

*stackaddr*线程属性被定义为栈的内存单元的最低地址，但这并不必然是栈的开始位置。对于某些处理器结构来说，栈是从高地址向低地址方向伸展的，那么*stackaddr*线程属性就是栈的结尾而不是开始位置。

391

pthread_attr_getstackaddr和pthread_attr_setstackaddr的缺陷在于*stackaddr*参数并没有明确地指定。它可以解释为栈的开始地址、还可以解释成用作栈的内存范围的最低地址。在栈内存地址空间从高地址向低地址扩展的处理器结构中，如果*stackaddr*参数是栈地址空间的最低地址，那么就需要知道栈的大小才能确定栈的开始位置。pthread_attr_getstack和pthread_attr_setstack函数纠正了这些缺陷。

应用程序也可以通过pthread_attr_getstacksize和pthread_attr_setstacksize函数读取或设置线程属性*stacksize*。

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

如果希望改变栈的默认大小，但又不想自己处理线程栈的分配问题，这时使用pthread_attr_setstacksize函数就非常有用。

线程属性*guardsize*控制着线程栈末尾之后用以避免栈溢出的扩展内存的大小。这个属性默认设置为PAGESIZE个字节。可以把*guardsize*线程属性设为0，从而不允许属性的这种特征行为发生：在这种情况下不会提供警戒缓冲区。同样地，如果对线程属性*stackaddr*作了修改，系统就会假设我们会自己管理栈，并使警戒栈缓冲区机制无效，等同于把*guardsize*线程属性设为0。

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

如果*guardsize*线程属性被修改了，操作系统可能把它取为页大小的整数倍。如果线程的栈指针溢出到警戒区域，应用程序就可能通过信号接收到出错信息。

Single UNIX Specification还定义了其他的一些可选的线程属性作为实时线程可选属性的一部分，但在这里不讨论这些属性。

更多的线程属性

线程还有其他的一些属性，这些属性并没有在pthread_attr_t结构中表达：

- 可取消状态（在12.7节中讨论）。
- 可取消类型（同样在12.7节中讨论）。
- 并发度。

392

并发度控制着用户级线程可以映射的内核线程或进程的数目。如果操作系统的实现在内核级的线程和用户级的线程之间保持一对一的映射，那么改变并发度并不会有什么效果，因为所有的用户级的线程都可能被调度到。但是，如果操作系统的实现让用户级线程到内核级线程或进程之间的映射关系是多对一的话，那么在给定时间内增加可运行的用户级线程数，可能会改善性能。pthread_setconcurrency函数可以用于提示系统，表明希望的并发度。

```
#include <pthread.h>

int pthread_getconcurrency(void);
```

返回值：当前的并发度

```
int pthread_setconcurrency(int level);
```

返回值：若成功则返回0，否则返回错误编号

pthread_getconcurrency函数返回当前的并发度。如果操作系统当前正控制着并发度（即之前没有调用过pthread_setconcurrency函数），那么pthread_getconcurrency

将返回0。

pthread_setconcurrency函数设定的并发度只是对系统的一个提示，系统并不保证请求的并发度一定会被采用。如果希望系统自己决定使用什么样的并发度，就把传入的参数*level*设为0。这样，应用程序调用*level*参数为0的pthread_setconcurrency函数，就可以撤销在这之前*level*参数非零的pthread_setconcurrency调用所产生的作用。

12.4 同步属性

就像线程具有属性一样，线程的同步对象也有属性。本节讨论互斥量、读写锁和条件变量的属性。

1. 互斥量属性

用pthread_mutexattr_init初始化pthread_mutexattr_t结构，用pthread_mutexattr_destroy来对该结构进行回收。

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

                              返回值：若成功则返回0，否则返回错误编号
```

393

pthread_mutexattr_init函数用默认的互斥量属性初始化pthread_mutexattr_t结构。值得注意的两个属性是进程共享属性和类型属性。POSIX.1中，进程共享属性是可选的，可以通过检查系统中是否定义了_POSIX_THREAD_PROCESS_SHARED符号来判断这个平台是否支持进程共享这个属性，也可以在运行时把_SC_THREAD_PROCESS_SHARED参数传给sysconf函数进行检查。虽然这个选项并不是遵循POSIX标准的操作系统必须提供的，但是Single UNIX Specification要求遵循XSI标准的操作系统支持这个选项。

在进程中，多个线程可以访问同一个同步对象。在第11章中已说明，这是默认的行为。在这种情况下，进程共享互斥量属性需设置为PTHREAD_PROCESS_PRIVATE。

第14章和第15章将说明，存在这样的机制，允许相互独立的多个进程把同一个内存区域映射到它们各自独立的地址空间中。就像多个线程访问共享数据一样，多个进程访问共享数据通常也需要同步。如果进程共享互斥量属性设置为PTHREAD_PROCESS_SHARED，从多个进程共享的内存区域中分配的互斥量就可以用于这些进程的同步。

可以使用pthread_mutexattr_getpshared函数查询pthread_mutexattr_t结构，得到它的进程共享属性，可以用pthread_mutexattr_setpshared函数修改进程共享属性。

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
                                 restrict attr,
                                 int *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);

                              返回值：若成功则返回0，否则返回错误编号
```

进程共享互斥量属性设为PTHREAD_PROCESS_PRIVATE时，允许pthread线程库提供更加有

效的互斥量实现，这在多线程应用程序中是默认的情况。在多个进程共享多个互斥量的情况下，pthread线程库可以限制开销较大的互斥量实现。

类型互斥量属性控制着互斥量的特性。POSIX.1定义了四种类型。PTHREAD_MUTEX_NORMAL类型是标准的互斥量类型，并不做任何特殊的错误检查或死锁检测。PTHREAD_MUTEX_ERRORCHECK互斥量类型提供错误检查。

PTHREAD_MUTEX_RECURSIVE互斥量类型允许同一线程在互斥量解锁之前对该互斥量进行多次加锁。用一个递归互斥量维护锁的计数，在解锁的次数和加锁次数不相同的情况下不会释放锁。所以如果对一个递归互斥量加锁两次，然后对它解锁一次，这个互斥量依然处于加锁状态，在对它再次解锁以前不能释放该锁。

394

最后，PTHREAD_MUTEX_DEFAULT类型可以用于请求默认语义。操作系统在实现它的时候可以把这种类型自由地映射到其他类型。例如，在Linux中，这种类型映射为普通的互斥量类型。

四种类型的行为如表12-4所示。“不占用时解锁”这一栏指的是一个线程对被另一个线程加锁的互斥量进行解锁的情况；“在已解锁时解锁”这一栏指的是当一个线程对已经解锁的互斥量进行解锁时将会发生的情况，这通常是编码错误所致。

表12-4 互斥量类型行为

互斥量类型	没有解锁时再次加锁?	不占用时解锁?	在已解锁时解锁?
PTHREAD_MUTEX_NORMAL	死锁	未定义	未定义
PTHREAD_MUTEX_ERRORCHECK	返回错误	返回错误	返回错误
PTHREAD_MUTEX_RECURSIVE	允许	返回错误	返回错误
PTHREAD_MUTEX_DEFAULT	未定义	未定义	未定义

可以用pthread_mutexattr_gettype函数得到互斥量类型属性，用pthread_mutexattr_settype函数修改互斥量类型属性。

```
#include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t *
                              restrict attr, int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
                    两者的返回值都是：若成功则返回0，否则返回错误编号
```

回忆11.6节中学过的，互斥量用于保护与条件变量关联的条件。在阻塞线程之前，pthread_cond_wait和pthread_cond_timedwait函数释放与条件相关的互斥量，这就允许其他线程获取互斥量、改变条件、释放互斥量并向条件变量发送信号。既然改变条件时必须占有互斥量，所以使用递归互斥量并不是好的办法。如果递归互斥量被多次加锁，然后用在调用pthread_cond_wait函数中，那么条件永远都不会得到满足，因为pthread_cond_wait所做的解锁操作并不能释放互斥量。

如果需要把现有的单线程接口放到多线程环境中，递归互斥量是非常有用的，但由于程序兼容性的限制，不能对函数接口进行修改。然而由于递归锁的使用需要一定技巧，它只应在没有其他可行方案的情况下使用。



图12-1解释了递归锁看似解决并发问题的情况。假设func1和func2是函数库中现有的函

395 数，其接口不能改变，因为存在调用这两个接口的应用程序，而且应用程序不能改动。

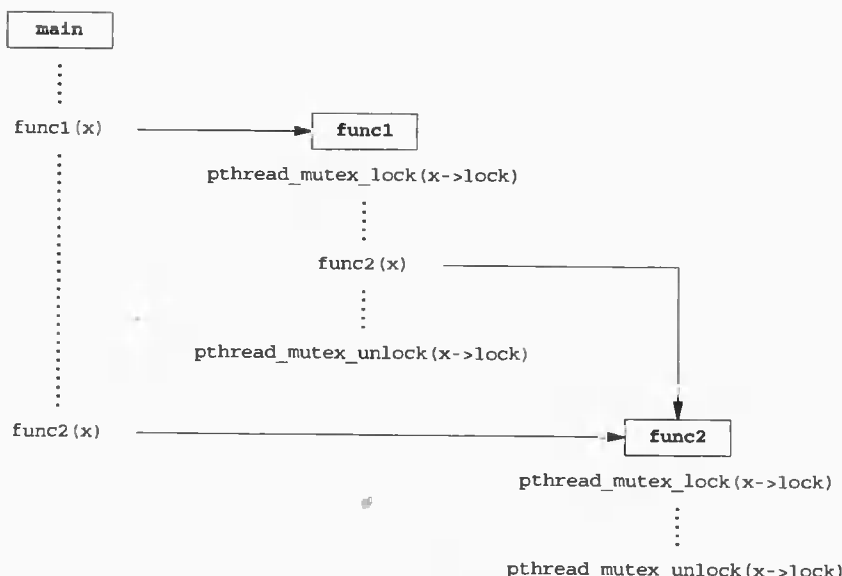


图12-1 使用递归锁的情况

为了保持接口跟原来相同，可以把互斥量嵌入到数据结构中，把这个数据结构的地址（x）作为参数传入。这种方案只有在为该数据结构提供了分配函数时才可行，所以应用并不知道数据结构的大小（假设在其中增加互斥量后必须扩大该数据结构的大小）。

> 如果在最初定义数据结构时，预留了足够的可填充字段，允许把一些填充字段替换成互斥量，那么这种方法也是可行的。不过，大多数程序员并不善于预测未来，所以这不是普遍可行的经验。

如果func1和func2函数都必须操作这个结构，而且可能会有多个线程同时访问该数据结构，那么func1和func2必须在操作数据以前对互斥量加锁。当func1必须调用func2时，如果互斥量不是递归类型，那么就会出现死锁。如果能在调用func2之前释放互斥量，在func2返回后重新获取互斥量，那么就可以避免使用递归互斥量，但这也给其他的线程提供了机会，其他的线程可能在func1执行期间得到互斥量的控制权，修改这个数据结构。这也许是不可接受的，当然具体的情况要取决于互斥量试图提供什么样的保护。

图12-2显示了这种情况下使用递归互斥量的另一种替代方法。通过提供func2函数的私有版本（称之为func2_locked函数），可以保持func1和func2函数接口不变，并且避免使用递归互斥量。要调用func2_locked函数，必须占有嵌入到数据结构中的互斥量，这个数据结构的地址是作为参数传入的。func2_locked的函数体包含func2的副本，func2现在只是用以获取互斥量，调用func2_locked，最后释放互斥量。 396

如果并不一定要保持库函数接口不变，就可以在每个函数中另外再加一个参数，以表明这个结构是否被调用者锁定。但是，如果可能的话，保持接口不变通常是更好的选择，这样可以避免实现过程中人为加入的东西对原有接口产生不良影响。

提供函数的加锁版本和不加锁版本，这样的策略在简单的情况下通常是可行的。在比较复杂的情况下，例如库需要调用库以外的函数，而且可能会再次回调库中的函数时，就需要依赖递归锁。□

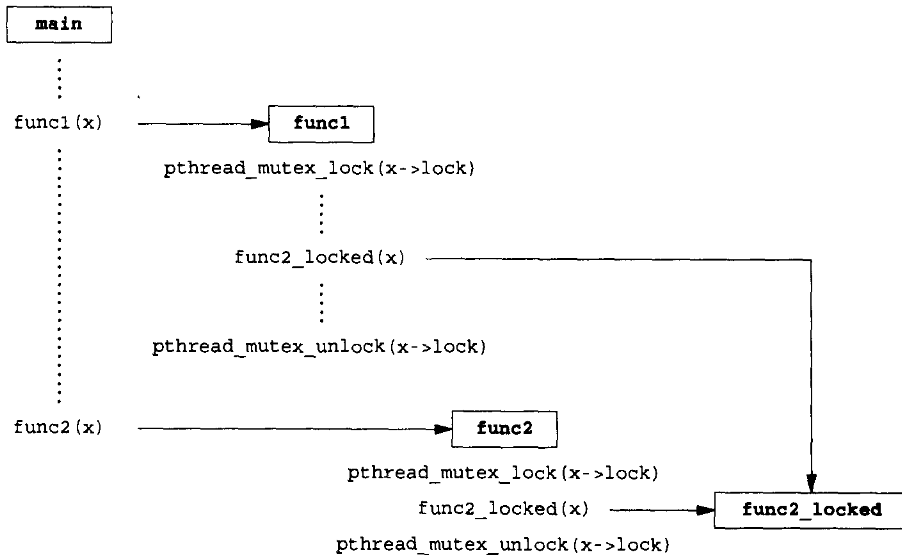


图12-2 避免使用递归锁的情况

实例

程序清单12-2解释了有必要使用递归互斥量的另一种情况。这里，有一个“超时”(timeout)函数，它允许另一个函数可以安排在未来的某个时间运行。假设线程并不是很昂贵的资源，可以为每个未决的超时函数创建一个线程。线程在时间未到时将一直等待，时间到了以后就调用请求的函数。

程序清单12-2 使用递归互斥量

```
#include "apue.h"
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

extern int makethread(void *(*)(void *), void *);

struct to_info {
    void    (*to_fn)(void *);    /* function */
    void    *to_arg;             /* argument */
    struct timespec to_wait;     /* time to wait */
};

#define SECTONSEC  1000000000    /* seconds to nanoseconds */
#define USECTONSEC 1000          /* microseconds to nanoseconds */

void *
timeout_helper(void *arg)
{
    struct to_info  *tip;

    tip = (struct to_info *)arg;
    nanosleep(&tip->to_wait, NULL);
    (*tip->to_fn)(tip->to_arg);
    return(0);
}
```

```
void
timeout(const struct timespec *when, void (*func)(void *), void *arg)
{
    struct timespec now;
    struct timeval  tv;
    struct to_info  *tip;
    int             err;

    gettimeofday(&tv, NULL);
    now.tv_sec = tv.tv_sec;
    now.tv_nsec = tv.tv_usec * USECTONSEC;
    if ((when->tv_sec > now.tv_sec) ||
      (when->tv_sec == now.tv_sec && when->tv_nsec > now.tv_nsec)) {
        tip = malloc(sizeof(struct to_info));
        if (tip != NULL) {
            tip->to_fn = func;
            tip->to_arg = arg;
            tip->to_wait.tv_sec = when->tv_sec - now.tv_sec;
            if (when->tv_nsec >= now.tv_nsec) {
                tip->to_wait.tv_nsec = when->tv_nsec - now.tv_nsec;
            } else {
                tip->to_wait.tv_sec--;
                tip->to_wait.tv_nsec = SECTONSEC - now.tv_nsec +
                  when->tv_nsec;
            }
            err = makethread(timeout_helper, (void *)tip);
            if (err == 0)
                return;
        }
    }

    /*
     * We get here if (a) when <= now, or (b) malloc fails, or
     * (c) we can't make a thread, so we just call the function now.
     */
    (*func)(arg);
}

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void
retry(void *arg)
{
    pthread_mutex_lock(&mutex);
    /* perform retry steps ... */
    pthread_mutex_unlock(&mutex);
}

int
main(void)
{
    int             err, condition, arg;
    struct timespec when;

    if ((err = pthread_mutexattr_init(&attr)) != 0)
        err_exit(err, "pthread_mutexattr_init failed");
    if ((err = pthread_mutexattr_settype(&attr,
      PTHREAD_MUTEX_RECURSIVE)) != 0)
        err_exit(err, "can't set recursive type");
    if ((err = pthread_mutex_init(&mutex, &attr)) != 0)
```

```
        err_exit(err, "can't create recursive mutex");
    /* ... */
    pthread_mutex_lock(&mutex);
    /* ... */
    if (condition) {
        /* calculate target time "when" */
        timeout(&when, retry, (void *)arg);
    }
    /* ... */
    pthread_mutex_unlock(&mutex);
    /* ... */
    exit(0);
}
```

如果不能创建线程，或者安排函数运行的时间已过，问题就出现了。在这种情况下，要从当前环境中调用之前请求运行的函数，因为函数要获取的锁和现在占有的锁是同一个，除非该锁是递归的，否则就会出现死锁。

397 ~ 399

这里使用程序清单12-1中的makethread函数以分离状态创建线程。希望函数在未来的某个时间运行，而且不希望一直等待线程结束。

可以调用sleep等待超时到达，但它提供的时间粒度是秒级的，如果希望等待的时间不是整数秒，需要用nanosleep(2)函数，它提供了类似的功能。

> 虽然nanosleep只有在Single UNIX Specification实时扩展中是必须实现的，但本文讨论的所有平台都支持该函数。

timeout的调用者需要占有互斥量来检查条件，并且把retry函数安排为原子操作。retry函数试图对同一个互斥量进行加锁，因此，除非互斥量是递归的，否则如果timeout函数直接调用retry就会导致死锁。 □

2. 读写锁属性

读写锁与互斥量类似，也具有属性。用pthread_rwlockattr_init初始化pthread_rwlockattr_t结构，用pthread_rwlockattr_destroy回收结构。

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

读写锁支持的唯一属性是进程共享属性，该属性与互斥量的进程共享属性相同。就像互斥量的进程共享属性一样，用一对函数来读取和设置读写锁的进程共享属性。

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
                                  restrict attr,
                                  int *restrict pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

虽然POSIX只定义了一个读写锁属性，但不同平台的实现可以自由地定义额外的、非标准的属性。

400

3. 条件变量属性

条件变量也有属性。与互斥量和读写锁类似，有一对函数用于初始化和回收条件变量属性。

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

与其他的同步原语一样，条件变量支持*进程共享*属性。

```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *
                                restrict attr,
                                int *restrict pshared);

int pthread_condattr_setpshared(pthread_condattr_t *attr,
                                int pshared);
```

两者的返回值都是：若成功则返回0，否则返回错误编号

12.5 重入

在10.6节中讨论了可重入函数和信号处理程序。在遇到重入问题时线程与信号处理程序类似。有了信号处理程序和线程，多个控制线程在同一时间可能潜在地调用同一个函数。

如果一个函数在同一时刻可以被多个线程安全地调用，就称该函数是*线程安全的*。在Single UNIX Specification中定义的所有函数，除了表12-5中列出的函数以外，其他函数都保证是线程安全的。另外，ctermid和tmpnam函数在参数传入空指针时并不能保证是线程安全的。类似地，wcrtomb和wcsrtombs函数如果参数mbstate_t传入的是空指针的话，也不能保证它们是线程安全的。

表12-5 POSIX.1中不能保证线程安全的函数

asctime	ecvt	gethostent	getutxline	putc_unlocked
basename	encrypt	getlogin	gmtime	putchar_unlocked
catgets	endgrent	getnetbyaddr	hcreate	putenv
crypt	endpwent	getnetbyname	hdestroy	pututxline
ctime	endutxent	getnetent	hsearch	rand
dbm_clearerr	fcvt	getopt	inet_ntoa	readdir
dbm_close	ftw	getprotobyname	l64a	setenv
dbm_delete	gcvt	getprotobynumber	lgamma	setgrent
dbm_error	getc_unlocked	getprotoent	lgammaf	setkey
dbm_fetch	getchar_unlocked	getpwent	lgammal	setpwent
dbm_firstkey	getdate	getpwnam	localeconv	setutxent
dbm_nextkey	getenv	getpwuid	localtime	strerror
dbm_open	getgrent	getservbyname	lrand48	strtok
dbm_store	getgrgid	getservbyport	mrand48	ttyname
dirname	getgrnam	getservent	nftw	unsetenv
dlerror	gethostbyaddr	getutxent	nl_langinfo	wcstombs
drand48	gethostbyname	getutxid	ptsname	wctomb

支持线程安全函数的操作系统实现会在<unistd.h>中定义符号_POSIX_THREAD_SAFE_FUNCTIONS。应用程序可以在sysconf函数中传入_SC_THREAD_SAFE_FUNCTIONS参数，以在运行时检查是否支持线程安全函数。所有遵循XSI的实现要求必须支持线程安全函数。

操作系统实现支持线程安全函数这一特性时，对POSIX.1中的一些非线程安全函数，它会提供可替代的线程安全版本，表12-6列出了这些函数的线程安全版本。很多函数并不是线程安全的，因为他们返回的数据是存放在静态的内存缓冲区中。通过修改接口，要求调用者自己提供缓冲区可以使函数变为线程安全的。

401

表12-6 替代的线程安全函数

acstime_r	gmtime_r
ctime_r	localtime_r
getgrgid_r	rand_r
getgrnam_r	readdir_r
getlogin_r	strerror_r
getpwnam_r	strtok_r
getpwuid_r	ttyname_r

表12-6中列出的函数的命名方式与他们的非线程安全版本的名字相似，只不过在名字最后加了_r，以表明这个版本是可重入的。

如果一个函数对多个线程来说是可重入的，则说这个函数是线程安全的，但这并不能说明对信号处理程序来说该函数也是可重入的。如果函数对异步信号处理程序的重入是安全的，那么就可以说函数是*异步－信号安全*的。在10.6节中讨论可重入函数时，表10-3中的函数就是异步信号安全函数。

除了表12-6中列出的函数，POSIX.1还提供了以线程安全的方式管理FILE对象的方法。可以使用flockfile和ftrylockfile获取与给定FILE对象关联的锁。这个锁是递归的，当占有这把锁的时候，还可以再次获取该锁，这并不会导致死锁。虽然这种锁的具体实现并无规定，但要求所有操作FILE对象的标准I/O例程表现得就像它们内部调用了flockfile和funlockfile一样。

402

```
#include <stdio.h>

int ftrylockfile(FILE *fp);

                                        返回值：若成功则返回0，否则返回非0值

void flockfile(FILE *fp);

void funlockfile(FILE *fp);
```

虽然标准的I/O例程从它们各自的内部数据结构这一角度出发，可能是以线程安全的方式实现的，但有时把锁开放给应用程序仍然是非常有用的。这允许应用程序把多个对标准I/O函数的调用组合成原子序列。当然，在处理多个FILE对象时，需要注意可能出现的死锁，并且需要对所有的锁仔细地排序。

如果标准I/O例程都获取它们各自的锁，那么在做一次一个字符的I/O操作时性能就会出现严重的下降。在这种情况下，需要对每一个字符的读或写操作进行获取锁和释放锁的动作。为了避免这种开销，出现了不加锁版本的基于字符的标准I/O例程。

```
#include <stdio.h>

int getchar_unlocked(void);

int getc_unlocked(FILE *fp);
```

两者的返回值都是：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF

```
int putchar_unlocked(int c);

int putc_unlocked(int c, FILE *fp);
```

两者的返回值都是：若成功则返回c，若出错则返回EOF

除非被flockfile（或ftrylockfile）和funlockfile的调用包围，否则尽量不要调用这四个函数，因为它们会导致不可预期的结果（即由多个控制线程非同步地访问数据所引起的种种问题）。

一旦对FILE对象进行加锁，就可以在释放锁之前对这些函数进行多次调用。这样就可以在多次的数据读写上分摊总的加解锁的开销。

程序清单12-3显示了getenv（见7.9节）一个可能的实现。因为所有调用getenv的线程返回的字符串都存放在同一个静态缓冲区中，所以这个版本不是可重入的。如果两个线程同时调用这个函数，就会看到不一致的结果。

403

程序清单12-3 getenv的非可重入版本

```
#include <limits.h>
#include <string.h>

static char envbuf[ARG_MAX];

extern char **environ;

char *
getenv(const char *name)
{
    int i, len;

    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            strcpy(envbuf, &environ[i][len+1]);
            return(envbuf);
        }
    }
    return(NULL);
}
```

程序清单12-4给出了genenv的可重入版本，这个版本命名为getenv_r。它使用pthread_once函数（在12.6节中描述）来确保每个进程只调用一次thread_init函数。

程序清单12-4 getenv的可重入（线程安全）版本

```
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>

extern char **environ;

pthread_mutex_t env_mutex;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;

static void
thread_init(void)
{
    pthread_mutexattr_t attr;

    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&env_mutex, &attr);
    pthread_mutexattr_destroy(&attr);
}
int
getenv_r(const char *name, char *buf, int buflen)
{
    int i, len, olen;

    pthread_once(&init_done, thread_init);
    len = strlen(name);
    pthread_mutex_lock(&env_mutex);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            olen = strlen(&environ[i][len+1]);
            if (olen >= buflen) {
                pthread_mutex_unlock(&env_mutex);
                return(ENOSPC);
            }
            strcpy(buf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex);
            return(0);
        }
    }
    pthread_mutex_unlock(&env_mutex);
    return(ENOENT);
}
```

404

要使getenv_r可重入，需要改变接口，调用者必须自己提供缓冲区，这样每个线程可以使用各自不同的缓冲区从而避免其他线程的干扰。但是注意这还不足以使getenv_r成为线程安全的，要使getenv_r成为线程安全的，需要在搜索请求的字符串时保护环境不被修改。我们可以使用互斥量，通过getenv_r和putenv函数对环境列表的访问进行序列化。

可以使用读写锁，从而允许对getenv_r的多次并发访问，但并发性的增强可能并不会在很大程度上改善程序的性能。这里面有两个原因：首先，环境列表通常不会很长，所以扫描列表时并不需要长时间地占有互斥量；其次，对getenv和putenv的调用不是频繁发生的，所以改善它们的性能并不会对程序的整体性能产生很大的影响。

即使把getenv_r变成线程安全的，也并不意味着它对信号处理程序是可重入的。如果使

用的是非递归的互斥量，当线程从信号处理程序中调用getenv_r时，就有可能出现死锁。如果信号处理程序在线程执行getenv_r时中断了该线程，由于这时已经占有加锁的env_mutex，这样其他线程试图对这个互斥量的加锁就会被阻塞，最终导致线程进入死锁状态。所以，必须使用递归互斥量阻止其他线程改变当前正查看的数据结构，同时还要阻止来自信号处理程序的死锁。问题是pthread函数并不保证是异步信号安全的，所以不能把pthread函数用于其他函数，让该函数成为异步信号安全的。 □

405

12.6 线程私有数据

线程私有数据（也称线程特定数据）是存储和查询与某个线程相关的数据的一种机制。把这种数据称为线程私有数据或线程特定数据的原因是，希望每个线程可以独立地访问数据副本，而不需要担心与其他线程的同步访问问题。

线程模型促进了进程中数据和属性的共享，许多人在设计线程模型时会遇到各种麻烦。但在这样的模型中，为什么还需要提出一些合适的用于阻止共享的接口呢？其中有两个原因：

第一，有时候需要维护基于每个线程的数据。因为线程ID并不能保证是小而连续的整数，所以不能简单地分配一个线程数据数组，用线程ID作为数组的索引。即使线程ID确实是小而连续的整数，可能还希望有一些额外的保护，以防止某个线程的数据与其他线程的数据相混淆。

采用线程私有数据的第二个原因是：它提供了让基于进程的接口适应多线程环境的机制。一个很明显的实例就是errno。回忆1.7节中对errno的讨论，（线程出现）以前的接口把errno定义为进程环境中全局可访问的整数。系统调用和库例程在调用或执行失败时设置errno，把它作为操作失败时的附属结果。为了让线程也能够使用那些原本基于进程的系统调用和库例程，errno被重新定义为线程私有数据。这样，一个线程做了设置errno的操作并不会影响进程中其他线程的errno值。

回顾前面所学可知，进程中的所有线程都可以访问进程的整个地址空间。除了使用寄存器以外，线程没有办法阻止其他线程访问它的数据，线程私有数据也不例外。虽然底层的实现部分并不能阻止这种访问能力，但管理线程私有数据的函数可以提高线程间的数据独立性。

在分配线程私有数据之前，需要创建与该数据关联的键。这个键将用于获取对线程私有数据的访问权。使用pthread_key_create创建一个键。

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *keyp,
                       void (*destructor)(void *));
```

返回值：若成功则返回0，否则返回错误编号

创建的键存放在*keyp*指向的内存单元，这个键可以被进程中的所有线程使用，但每个线程把这个键与不同的线程私有数据地址进行关联。创建新键时，每个线程的数据地址设为null值。

除了创建键以外，pthread_key_create可以选择为该键关联析构函数，当线程退出时，如果数据地址已经被置为非null数值，那么析构函数就会被调用，它唯一的参数就是该数据地址。如果传入的*destructor*参数为null，就表明没有析构函数与键关联。当线程调用pthread_exit或者线程执行返回，正常退出时，析构函数就会被调用，但如果线程调用了exit、_exit、_Exit、abort或出现其他非正常的退出时，就不会调用析构函数。

406

线程通常使用malloc为线程私有数据分配内存空间，析构函数通常释放已分配的内存。

如果线程没有释放内存就退出了，那么这块内存将会丢失，即线程所属进程出现了内存泄漏。

线程可以为线程私有数据分配多个键，每个键都可以有一个析构函数与它关联。各个键的析构函数可以互不相同，当然它们也可以使用相同的析构函数。每个操作系统在实现的时候可以对进程可分配的键的数量进行限制（回忆表12-1中的PTHREAD_KEYS_MAX）。

线程退出时，线程私有数据的析构函数将按照操作系统实现中定义的顺序被调用。析构函数可能会调用另一个函数，该函数可能会创建新的线程私有数据而且把这个数据与当前的键关联起来。当所有的析构函数都调用完成以后，系统会检查是否还有非null的线程私有数据值与键关联，如果有的话，再次调用析构函数。这个过程将会一直重复直到线程所有的键都为null值线程私有数据，或者已经做了PTHREAD_DESTRUCTOR_ITERATIONS（见表12-1）中定义的最大次数的尝试。

对所有的线程，都可以通过调用pthread_key_delete来取消键与线程私有数据值之间的关联关系。

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t *key);
```

返回值：若成功则返回0，否则返回错误编号

注意调用pthread_key_delete并不会激活与键关联的析构函数。要释放任何与键对应的线程私有数据值的内存空间，需要在应用程序中采取额外的步骤。

需要确保分配的键并不会由于在初始化阶段的竞争而发生变动。下列代码可以导致两个线程都调用pthread_key_create：

```
void destructor(void *);

pthread_key_t key;
int init_done = 0;

int
threadfunc(void *arg)
{
	if (!init_done) {
		init_done = 1;
		err = pthread_key_create(&key, destructor);
	}
	...
}
```

407

有些线程可能看到某个键值，而其他的线程看到的可能是另一个不同的键值，这取决于系统是如何调度线程的，解决这种竞争的办法是使用pthread_once。

```
#include <pthread.h>

pthread_once_t initflag = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *initflag, void (*initfn)(void));
```

返回值：若成功则返回0，否则返回错误编号

*initflag*必须是一个非本地变量（即全局变量或静态变量），而且必须初始化为PTHREAD_ONCE_INIT。

如果每个线程都调用pthread_once，系统就能保证初始化例程*initfn*只被调用一次，即在系统首次调用pthread_once时。创建键时避免出现竞争的一个恰当的方法可以描述如下：

```
void destructor(void *);

pthread_key_t key;
pthread_once_t init_done = PTHREAD_ONCE_INIT;

void
thread_init(void)
{
    err = pthread_key_create(&key, destructor);
}

int
threadfunc(void *arg)
{
    pthread_once(&init_done, thread_init);
    ...
}
```

键一旦创建，就可以通过调用pthread_setspecific函数把键和线程私有数据关联起来。可以通过pthread_getspecific函数获得线程私有数据的地址。

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
```

返回值：线程私有数据值；若没有值与键关联则返回NULL

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

返回值：若成功则返回0，否则返回错误编号

408

如果没有线程私有数据值与键关联，pthread_getspecific将返回一个空指针，可以据此来确定是否需要调用pthread_setspecific。

实例

在程序清单12-3中，给出了getenv的假设实现，接着又给出了一个新的接口，提供的功能相同，不过它是线程安全的（见程序清单12-4）。但是如果无法修改应用程序以直接使用新的接口会出现什么问题呢？这种情况下，可以使用线程私有数据来维护每个线程的数据缓冲区的副本，用于存放各自的返回字符串，如程序清单12-5所示。

程序清单12-5　线程安全的getenv的兼容版本

```
#include <limits.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>

static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex = PTHREAD_MUTEX_INITIALIZER;

extern char **environ;
```

```
static void
thread_init(void)
{
    pthread_key_create(&key, free);
}

char *
getenv(const char *name)
{
    int     i, len;
    char    *envbuf;

    pthread_once(&init_done, thread_init);
    pthread_mutex_lock(&env_mutex);
    envbuf = (char *)pthread_getspecific(key);
    if (envbuf == NULL) {
        envbuf = malloc(ARG_MAX);
        if (envbuf == NULL) {
            pthread_mutex_unlock(&env_mutex);
            return(NULL);
        }
        pthread_setspecific(key, envbuf);
    }
    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            strcpy(envbuf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex);
            return(envbuf);
        }
    }
    pthread_mutex_unlock(&env_mutex);
    return(NULL);
}
```

409

使用pthread_once来确保只为将要使用的线程私有数据创建了一个键。如果pthread_getspecific返回的是空指针，需要分配内存然后把键与该内存单元关联，否则如果返回的不是空指针，就使用pthread_getspecific返回的内存单元。对析构函数，使用free来释放之前由malloc分配的内存。只有当线程私有数据值为非null时，析构函数才会被调用。

注意虽然这个版本的getenv是线程安全的，但它并不是异步-信号安全的。对信号处理程序而言，即使使用递归的互斥量，这个版本的getenv也不可能是可重入的，因为它调用了malloc，而malloc函数本身并不是异步-信号安全的。 □

12.7 取消选项

有两个线程属性并没有包含在pthread_attr_t结构中，它们是可取消状态和可取消类型。这两个属性影响着线程在响应pthread_cancel函数调用时所呈现的行为（见11.5节）。

可取消状态属性可以是PTHREAD_CANCEL_ENABLE，也可以是PTHREAD_CANCEL_DISABLE。线程可以通过调用pthread_setcancelstate修改它的可取消状态。

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

返回值：若成功则返回0，否则返回错误编号

pthread_setcancelstate把当前的可取消状态设置为*state*，把原来的可取消状态存放在由*oldstate*指向的内存单元中，这两步是原子操作。

回顾11.5节，pthread_cancel调用并不等待线程终止，在默认情况下，线程在取消请求发出以后还是继续运行，直到线程到达某个取消点。取消点是线程检查是否被取消并按照请求进行动作的一个位置。POSIX.1保证在线程调用表12-7中列出的任何函数时，取消点都会出现。

410

表12-7　POSIX.1定义的取消点

accept	mq_timedsend	putpmsg	sigsuspend
aio_suspend	msgrcv	pwrite	sigtimedwait
clock_nanosleep	msgsnd	read	sigwait
close	msync	readv	sigwaitinfo
connect	nanosleep	recv	sleep
creat	open	recvfrom	system
fcntl2	pause	recvmsg	tcdrain
fsync	poll	select	usleep
getmsg	pread	sem_timedwait	wait
getpmsg	pthread_cond_timedwait	sem_wait	waitid
lockf	pthread_cond_wait	send	waitpid
mq_receive	pthread_join	sendmsg	write
mq_send	pthread_testcancel	sendto	writev
mq_timedreceive	putmsg	sigpause	

线程启动时默认的可取消状态是PTHREAD_CANCEL_ENABLE。当状态设为PTHREAD_CANCEL_DISABLE时，对pthread_cancel的调用并不会杀死线程；相反，取消请求对这个线程来说处于未决状态。当取消状态再次变为PTHREAD_CANCEL_ENABLE时，线程将在下一个取消点上对所有未决的取消请求进行处理。

除了表12-7中列出的函数，POSIX.1还指定了表12-8中列出的函数作为可选的取消点。

> 注意表12-8中列出的有些函数并没有在本书中进一步讨论。许多函数在Single UNIX Specification中是可选的。

如果应用程序在很长一段时间内都不会调用到表12-7或表12-8中的函数（例如计算数学领域的应用程序），那么可以调用pthread_testcancel函数在程序中自己添加取消点。

```
#include <pthread.h>

void pthread_testcancel(void);
```

调用pthread_testcancel时，如果有某个取消请求正处于未决状态，而且取消并没有置为无效，那么线程就会被取消。但是如果取消被置为无效时，pthread_testcancel调用就没有任何效果。

这里所描述的默认取消类型也称为延迟取消。调用pthread_cancel以后，在线程到达取消点之前，并不会出现真正的取消。可以通过调用pthread_setcanceltype来修改取消类型。

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);

                                  返回值：若成功则返回0，否则返回错误编号
```

411

表12-8 POSIX.1定义的可选取消点

catclose	ftell	getwc	printf
catgets	ftello	getwchar	putc
catopen	ftw	getwd	putc_unlocked
closedir	fwprintf	glob	putchar
closelog	fwrite	iconv_close	putchar_unlocked
ctermid	fwscanf	iconv_open	puts
dbm_close	getc	ioctl	pututxline
dbm_delete	getc_unlocked	lseek	putwc
dbm_fetch	getchar	mkstemp	putwchar
dbm_nextkey	getchar_unlocked	nftw	readdir
dbm_open	getcwd	opendir	readdir_r
dbm_store	getdate	openlog	remove
dlclose	getgrent	pclose	rename
dlopen	getgrgid	perror	rewind
endgrent	getgrgid_r	popen	rewinddir
endhostent	getgrnam	posix_fadvise	scanf
endnetent	getgrnam_r	posix_fallocate	seekdir
endprotoent	gethostbyaddr	posix_madvise	semop
endpwent	gethostbyname	posix_spawn	setgrent
endservent	gethostent	posix_spawnp	sethostent
endutxent	gethostname	posix_trace_clear	setnetent
fclose	getlogin	posix_trace_close	setprotoent
fcntl	getlogin_r	posix_trace_create	setpwent
fflush	getnetbyaddr	posix_trace_create_withlog	setservent
fgetc	getnetbyname	posix_trace_eventtypelist_getnext_id	setutxent
fgetpos	getnetent	posix_trace_eventtypelist_rewind	strerror
fgets	getprotobyname	posix_trace_flush	syslog
fgetwc	getprotobynumber	posix_trace_get_attr	tmpfile
fgetws	getprotoent	posix_trace_get_filter	tmpnam
fopen	getpwent	posix_trace_get_status	ttyname
fprintf	getpwnam	posix_trace_getnext_event	ttyname_r
fputc	getpwnam_r	posix_trace_open	ungetc
fputs	getpwuid	posix_trace_rewind	ungetwc
fputwc	getpwuid_r	posix_trace_set_filter	unlink
fputws	gets	posix_trace_shutdown	vfprintf
fread	getservbyname	posix_trace_timedgetnext_event	vfwprintf
freopen	getservbyport	posix_typed_mem_open	vprintf
fscanf	getservent	pthread_rwlock_rdlock	vwprintf
fseek	getutxent	pthread_rwlock_timedrdlock	wprintf
fseeko	getutxid	pthread_rwlock_timedwrlock	wscanf
fsetpos	getutxline	pthread_rwlock_wrlock	

*type*参数可以是PTHREAD_CANCEL_DEFERRED，也可以是PTHREAD_CANCEL_ASYNCHRONOUS，pthread_setcanceltype函数把取消类型设置为*type*，把原来的取消类型返回到*oldtype*指向的整型单元。

异步取消与延迟取消不同，使用异步取消时，线程可以在任意时间取消，而不是非得遇到取消点才能被取消。 412

12.8 线程和信号

即使是在基于进程的编程模式中，信号的处理也可能是很复杂的。把线程引入编程范型，就使信号的处理变得更加复杂。

每个线程都有自己的信号屏蔽字，但是信号的处理是进程中所有线程共享的。这意味着尽管单个线程可以阻止某些信号，但当线程修改了与某个信号相关的处理行为以后，所有的线程都必须共享这个处理行为的改变。这样如果一个线程选择忽略某个信号，而其他的线程可以恢复信号

的默认处理行为，或者为信号设置一个新的处理程序，从而可以撤销上述线程的信号选择。

进程中的信号是递送到单个线程的。如果信号与硬件故障或计时器超时相关，该信号就被发送到引起该事件的线程中去，而其他的信号则被发送到任意一个线程。

10.12节讨论了进程如何使用sigprocmask来阻止信号发送。sigprocmask的行为在多线程的进程中并没有定义，线程必须使用pthread_sigmask。

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

返回值：若成功则返回0，否则返回错误编号

pthread_sigmask函数与sigprocmask函数基本相同，除了pthread_sigmask工作在线程中，并且失败时返回错误码，而不像sigprocmask中那样设置errno并返回-1。

线程可以通过调用sigwait等待一个或多个信号发生。

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict signop);
```

返回值：若成功则返回0，否则返回错误编号

*set*参数指出了线程等待的信号集，*signop*指向的整数将作为返回值，表明发送信号的数量。

如果信号集中的某个信号在sigwait调用的时候处于未决状态，那么sigwait将无阻塞地返回，在返回之前，sigwait将从进程中移除那些处于未决状态的信号。为了避免错误动作发生，线程在调用sigwait之前，必须阻塞那些它正在等待的信号。sigwait函数会自动取消信号集的阻塞状态，直到有新的信号被递送。在返回之前，sigwait将恢复线程的信号屏蔽字。如果信号在sigwait调用的时候没有被阻塞，在完成对sigwait调用之前会出现一个时间窗，在这个窗口期，某个信号可能在线程完成sigwait调用之前就被递送了。

使用sigwait的好处在于它可以简化信号处理，允许把异步产生的信号用同步的方式处理。为了防止信号中断线程，可以把信号加到每个线程的信号屏蔽字中，然后安排专用线程作信号处理。这些专用线程可以进行函数调用，不需要担心在信号处理程序中调用哪些函数是安全的，因为这些函数调用来自正常的线程环境，而非传统的信号处理程序，传统信号处理程序通常会中断线程的正常执行。

413

如果多个线程在sigwait调用时，等待的是同一个信号，这时就会出现线程阻塞。当信号递送的时候，只有一个线程可以从sigwait中返回。如果信号被捕获（例如进程通过使用sigaction建立了一个信号处理程序），而且线程正在sigwait调用中等待同一信号，那么这时将由操作系统实现来决定以何种方式递送信号。在这种情况下，操作系统实现可以让sigwait返回，也可以激活信号处理程序，但不可能出现两者皆可的情况。

要把信号发送到进程，可以调用kill（见10.9节）；要把信号发送到线程，可以调用pthread_kill。

```
#include <signal.h>

int pthread_kill(pthread_t thread, int signo);
```

返回值：若成功则返回0，否则返回错误编号

可以传一个0值的*signo*来检查线程是否存在。如果信号的默认处理动作是终止该进程，那么把信号传递给某个线程仍然会杀掉整个进程。

注意闹钟定时器是进程资源，并且所有的线程共享相同的alarm。所以进程中的多个线程不可能互不干扰（或互不合作）地使用闹钟定时器（这是习题12.6的内容）。

回忆程序清单10-16，等待信号处理程序设置标志，从而表明主程序应该退出。唯一可运行的控制线程就是主线程和信号处理程序，所以阻塞信号足以避免错失标志修改。在线程中，需要使用互斥量来保护标志，如程序清单12-6所示。

程序清单12-6 同步信号处理

```
#include "apue.h"
#include <pthread.h>

int         quitflag;   /* set nonzero by thread */
sigset_t    mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wait = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "sigwait failed");
        switch (signo) {
        case SIGINT:
            printf("\ninterrupt\n");
            break;

        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&wait);
            return(0);

        default:
            printf("unexpected signal %d\n", signo);
            exit(1);
        }
    }
}

int
main(void)
{
    int         err;
    sigset_t    oldmask;
    pthread_t   tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
```

414

```
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "SIG_BLOCK error");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&wait, &lock);
    pthread_mutex_unlock(&lock);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;

    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

415

这里并不让信号处理程序中断主控线程，而是由专门的独立控制线程进行信号处理。改动quitflag的值是在互斥量的保护下进行的，这样主控线程不会在调用pthread_cond_signal时错失唤醒调用。在主控线程中使用相同的互斥量来检查标志的值，并且原子地释放互斥量，等待条件的发生。

注意在主线程开始时阻塞SIGINT和SIGQUIT。当创建线程进行信号处理时，新建线程继承了现有的信号屏蔽字。因为sigwait会解除信号的阻塞状态，所以只有一个线程可以用于信号的接收。这使得对主线程进行编码时不必担心来自这些信号的中断。

运行这个程序可以得到与程序清单10-16类似的输出结果：

```
$ ./a.out
^?                    键入中断字符
interrupt
^?                    再次键入中断字符
interrupt
^?                    再一次
interrupt
^\ $                  用结束字符终止
```

□

Linux线程是以独立进程实现的，使用clone(2)共享资源。因此Linux线程在遇到信号时的行为与其他操作系统实现不同。在POSIX.1线程模型中，异步信号被发送到进程以后，进程中当前没有阻塞该信号的某个线程就被选中，接收信号。在Linux上，异步信号发送到特定的线程，而且因为每个线程是作为独立的进程执行的，系统就不能选择当前没有阻塞该信号的线程。这样一来，结果就是线程可能不会注意到该信号。因此，如果信号产生于终端驱动程序，这样的信号是通知到进程组的，像程序清单12-6中的程序就可以工作；但是如果试图用kill把信号发送给进程时，在Linux上就不能如预期的那样工作。

12.9　线程和fork

当线程调用fork时，就为子进程创建了整个进程地址空间的副本。回忆8.3节中讨论的写时复制，子进程与父进程是完全不同的进程，只要两者都没有对内存做出改动，父进程和子进程之间还可以共享内存页的副本。

子进程通过继承整个地址空间的副本，也从父进程那里继承了所有互斥量、读写锁和条件变量的状态。如果父进程包含多个线程，子进程在fork返回以后，如果紧接着不是马上调用exec的话，就需要清理锁状态。

在子进程内部只存在一个线程，它是由父进程中调用fork的线程的副本构成的。如果父进程中的线程占有锁，子进程同样占有这些锁。问题是子进程并不包含占有锁的线程的副本，所以子进程没有办法知道它占有了哪些锁并且需要释放哪些锁。 416

如果子进程从fork返回以后马上调用某个exec函数，就可以避免这样的问题。这种情况下，老的地址空间被丢弃，所以锁的状态无关紧要。但如果子进程需要继续做处理工作的话，这种方法就行不通，还需要使用其他的策略。

要清除锁状态，可以通过调用pthread_atfork函数建立*fork处理程序*。

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
                   void (*child)(void));
```

返回值：若成功则返回0，否则返回错误编号

用pthread_atfork函数最多可以安装三个帮助清理锁的函数。*prepare* fork处理程序由父进程在fork创建子进程前调用，这个fork处理程序的任务是获取父进程定义的所有锁。*parent* fork处理程序是在fork创建了子进程以后，但在fork返回之前在父进程环境中调用的，这个fork处理程序的任务是对*prepare* fork处理程序获得的所有锁进行解锁。*child* fork处理程序在fork返回之前在子进程环境中调用，与*parent* fork处理程序一样，*child* fork处理程序也必须释放*prepare* fork处理程序获得的所有锁。

注意不会出现加锁一次解锁两次的情况，虽然看起来也许会出现。当子进程地址空间创建的时候，它得到了父进程定义的所有锁的副本。因为*prepare* fork处理程序获取所有的锁，父进程中的内存和子进程中的内存内容在开始的时候是相同的。当父进程和子进程对他们的锁的副本进行解锁的时候，新的内存是分配给子进程的，父进程的内存内容被复制到子进程的内存中（写时复制），所以就会陷入这样的假象，看起来父进程对它所有的锁的副本进行了加锁，子进程对它所有的锁的副本进行了加锁。父进程和子进程对在不同内存位置的重复的锁都进行了解锁操作，就好像出现了下列的事件序列：

(1) 父进程获得所有的锁。

(2) 子进程获得所有的锁。

(3) 父进程释放它的锁。

(4) 子进程释放它的锁。

可以多次调用pthread_atfork函数从而设置多套fork处理程序。如果不需要使用其中某个处理程序，可以给特定的处理程序参数传入空指针，这样它就不会起任何作用。使用多个fork处理程序时，处理程序的调用顺序并不相同。*parent*和*child* fork处理程序是以它们注册时的顺序进行调用的，而*prepare* fork处理程序的调用顺序与它们注册时的顺序相反。这样可以允许多个模块注册它们自己的fork处理程序，并且保持锁的层次。 417

例如，假设模块A调用模块B中的函数，而且每个模块有自己的一套锁。如果锁的层次是A在B之前，模块B必须在模块A之前设置fork处理程序。当父进程调用fork时，就会执行以下的步骤，假设子进程在父进程之前运行。

(1) 调用模块A的*prepare* fork处理程序获取模块A的所有锁。
(2) 调用模块B的*prepare* fork处理程序获取模块B的所有锁。
(3) 创建子进程。
(4) 调用模块B中的*child* fork处理程序释放子进程中模块B的所有锁。
(5) 调用模块A中的*child* fork处理程序释放子进程中模块A的所有锁。
(6) fork函数返回到子进程。
(7) 调用模块B中的*parent* fork处理程序释放父进程中模块B的所有锁。
(8) 调用模块A中*parent* fork处理程序来释放父进程中模块A的所有锁。
(9) fork函数返回到父进程。

如果fork处理程序是为了清理锁状态，那么又由谁来负责清理条件变量的状态呢？在有些操作系统的实现中，条件变量可能并不需要做任何清理。但是有些操作系统实现把锁作为条件变量实现的一部分，这种情况下的条件变量就需要清理。问题是目前不存在这样的接口允许做锁的清理工作，如果锁是嵌入到条件变量的数据结构中的，那么在调用fork之后就不能使用条件变量，因为还没有可移植的方法对锁进行状态清理。另外，如果操作系统的实现是使用全局锁保护进程中所有的条件变量数据结构，那么操作系统实现本身可以在fork库例程中做清理锁的工作，但是应用程序不应该依赖操作系统实现中这样的细节。

程序清单12-7中的程序描述了如何使用pthread_atfork和fork处理程序。

程序清单12-7 pthread_atfork实例

```
#include "apue.h"
#include <pthread.h>

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void
prepare(void)
{
    printf("preparing locks...\n");
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
}

void
parent(void)
{
    printf("parent unlocking locks...\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}

void
child(void)
{
    printf("child unlocking locks...\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}

void *
```

418

```
thr_fn(void *arg)
{
    printf("thread started...\n");
    pause();
    return(0);
}

int
main(void)
{
    int         err;
    pid_t       pid;
    pthread_t   tid;

#if defined(BSD) || defined(MACOS)
    printf("pthread_atfork is unsupported\n");
#else
    if ((err = pthread_atfork(prepare, parent, child)) != 0)
        err_exit(err, "can't install fork handlers");
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");
    sleep(2);
    printf("parent about to fork...\n");
    if ((pid = fork()) < 0)
        err_quit("fork failed");
    else if (pid == 0)  /* child */
        printf("child returned from fork\n");
    else        /* parent */
        printf("parent returned from fork\n");
#endif
    exit(0);
}
```

419

程序中定义了两个互斥量，lock1和lock2，*prepare* fork处理程序获取这两把锁，*child* fork处理程序在子进程环境中释放锁，*parent* fork处理程序在父进程中释放锁。

运行该程序，得到如下输出：

```
$ ./a.out
thread started...
parent about to fork...
preparing locks...
child unlocking locks...
child returned from fork
parent unlocking locks...
parent returned from fork
```

可以看出，*prepare* fork处理程序在调用fork以后运行，*child* fork处理程序在fork调用返回到子进程之前运行，*parent* fork处理程序在fork调用返回给父进程前运行。 □

12.10 线程和I/O

在3.11节中介绍了pread和pwrite函数，这些函数在多线程环境下是非常有帮助的，因为进程中的所有线程共享相同的文件描述符。

考虑两个线程，在同一时间对同一个文件描述符进行读写操作。

```
        线程 A                                    线程 B
lseek(fd, 300, SEEK_SET);                lseek(fd, 700, SEEK_SET);
read(fd, buf1, 100);                     read(fd, buf2, 100);
```

如果线程A执行lseek，然后线程B在线程A调用read之前调用lseek，那么两个线程最终会读取同一条记录。很显然这不是我们希望的。

为了解决这个问题，可以使用pread，使偏移量的设定和数据的读取成为一个原子操作。

```
        线程 A                                    线程 B
pread(fd, buf1, 100, 300);               pread(fd, buf2, 100, 700);
```

使用pread可以确保线程A读取偏移量为300的记录，而线程B读取偏移量为700的记录。可以使用pwrite来解决并发线程对同一文件进行写操作的问题。

12.11 小结

420 在UNIX系统中，线程提供了分解并发任务的一种替代模型。线程促进了独立控制线程之间的共享，但也带来了它特有的同步问题。本章中，我们考查了如何调整线程和它们的同步原语，讨论了线程的可重入性，还学习了线程如何与其他面向进程的系统调用进行交互。

习题

12.1 在Linux系统中运行程序清单12-7中的程序，但把输出结果重定向到文件中，并解释结果。

12.2 实现putenv_r，即putenv的可重入版本。确保实现既是线程安全的，也是异步信号安全的。

12.3 是否可以通过在函数开始的时候阻塞信号，并在函数返回之前恢复原来的信号屏蔽字这种方法，让程序清单12-5中的程序变成异步信号安全的？解释其原因。

12.4 写一个程序运用程序清单12-5中的getenv版本，在FreeBSD上编译并运行该程序，会出现什么结果？解释其原因。

12.5 假设可以在一个程序中创建多个线程执行不同的任务，为什么还是有可能会用fork？解释其原因。

12.6 重新实现程序清单10-21中的程序，在不使用nanosleep的情况下使它成为线程安全的。

12.7 调用fork以后，是否可以通过首先用pthread_cond_destroy销毁条件变量，然后用pthread_cond_init初始化条件变量这种方法，安全地在子进程中对条件变量进行重新初始化？

421

第13章

守护进程

13.1 引言

守护进程也称精灵进程（daemon）是生存期较长的一种进程。它们常常在系统自举时启动，仅在系统关闭时才终止。因为它们没有控制终端，所以说它们是在后台运行的。UNIX系统有很多守护进程，它们执行日常事务活动。

本章说明守护进程的结构，以及如何编写守护进程程序。因为守护进程没有控制终端，我们需要了解在出现问题时，守护进程如何报告出错情况。

有关术语daemon如何用于计算机系统的历史背景，参见Raymond[1996]。

13.2 守护进程的特征

先来查看一些常用的系统守护进程，以及它们怎样和第9章中所叙述的进程组、控制终端和会话等概念相关联。ps(1)命令打印系统中各个进程的状态。该命令有多个选项，有关细节请参考系统手册。为了解本节讨论中所需的信息，在基于BSD的系统下执行：

```
ps -axj
```

选项-a显示由其他用户所拥有的进程的状态。-x显示没有控制终端的进程状态。-j显示与作业有关的信息：会话ID、进程组ID、控制终端以及终端进程组ID。在基于系统V的系统中，与此相类似的命令是ps-efjc（为了提高安全性，某些UNIX系统不允许用户使用ps命令查看不属于自己的进程）。ps的输出大致是：

423

```
PPID   PID  PGID   SID TTY TPGID UID   COMMAND
   0     1     0     0 ?      -1   0   init
   1     2     1     1 ?      -1   0   [keventd]
   1     3     1     1 ?      -1   0   [kapmd]
   0     5     1     1 ?      -1   0   [kswapd]
   0     6     1     1 ?      -1   0   [bdflush]
   0     7     1     1 ?      -1   0   [kupdated]
   1  1009  1009  1009 ?      -1  32   portmap
   1  1048  1048  1048 ?      -1   0   syslogd -m 0
   1  1335  1335  1335 ?      -1   0   xinetd -pidfile /var/run/xinetd.pid
   1  1403     1     1 ?      -1   0   [nfsd]
   1  1405     1     1 ?      -1   0   [lockd]
1405  1406     1     1 ?      -1   0   [rpciod]
   1  1853  1853  1853 ?      -1   0   crond
   1  2182  2182  2182 ?      -1   0   /usr/sbin/cupsd
```

其中，已移去了一些我们并无兴趣的列，例如累计CPU时间。按照顺序，各列标题的意义是：

父进程ID、进程ID、进程组ID、会话ID、终端名称、终端进程组ID（与该控制终端相关的前台进程组）、用户ID以及命令字符串。

> 此ps命令在支持会话ID的系统（Linux）上运行，9.5节的setsid函数中曾提及会话ID。它是会话首进程的进程ID。但是，基于BSD的系统将打印与本进程所属进程组对应的session结构的地址（见9.11节）。

系统进程依赖于操作系统实现。父进程ID为0的各进程通常是内核进程，它们作为系统自举过程的一部分而启动。(init是此种进程的例外，它是内核在自举时启动的用户层命令。）内核进程是特殊的，通常存在于系统的整个生命期中。它们以超级用户特权运行，无控制终端，无命令行。

进程1通常是init，我们已在8.2节对此作过说明。它是一个系统守护进程，负责启动各运行层次特定的系统服务。这些服务通常是在它们自己拥有的守护进程的帮助下实现的。

在Linux下，keventd守护进程为在内核中运行计划执行的函数提供进程上下文。kapmd守护进程对很多计算机系统中具有的高级电源管理提供支持。kswapd守护进程也称为页面调出守护进程（pageout daemon)。它通过将脏页面以低速写到磁盘上从而使这些页面在需要时仍可回收使用，这种方式支持虚存子系统。

Linux内核使用两个守护进程bdflush和kupdated将高速缓存中的数据冲洗到磁盘上。当可用内存达到下限时，bdflush守护进程将脏缓冲区从缓冲池（buffer cache）中冲洗到磁盘上。每隔一定时间间隔，kupdated守护进程将脏页面冲洗到磁盘上，以便在系统失效时减少丢失的数据。

424

端口映射守护进程portmap提供将RPC（Remote Procedure Call, 远程过程调用）程序号映射为网络端口号的服务。syslogd守护进程可由帮助操作人员把系统消息记入日志的任何程序使用。可以在一台实际的控制台上打印这些消息，也可将它们写到一个文件中（13.4节将对syslog设施进行说明)。

在9.3节已谈到inetd守护进程（xinetd)。它侦听系统网络接口，以便取得来自网络的对各种网络服务进程的请求。nfsd、lockd和rpciod守护进程提供对网络文件系统(Network File System, NFS）的支持。

cron守护进程（crond）在指定的日期和时间执行指定的命令。许多系统管理任务是由cron定期地执行相关程序而实现的。cupsd守护进程是打印假脱机进程，它处理对系统提出的所有打印请求。

注意，大多数守护进程都以超级用户（用户ID为0）特权运行。没有一个守护进程具有控制终端，其终端名设置为问号（?），终端前台进程组ID设置为-1。内核守护进程以无控制终端方式启动。用户层守护进程缺少控制终端可能是守护进程调用了setsid的结果。所有用户层守护进程都是进程组的组长进程以及会话的首进程，而且是这些进程组和会话中的唯一进程。最后，应当引起注意的是大多数守护进程的父进程是init进程。

13.3 编程规则

在编写守护进程程序时需遵循一些基本规则，以便防止产生并不需要的交互作用。下面先说明这些规则，然后给出一个按照这些规则编写的函数daemonize。

(1) 首先要做的是调用umask将文件模式创建屏蔽字设置为0。由继承得来的文件模式创建

屏蔽字可能会拒绝设置某些权限。例如，若守护进程要创建一个组可读、写的文件，而继承的文件模式创建屏蔽字可能屏蔽了这两种权限，于是所要求的组可读、写就不能起作用。

(2) 调用fork，然后使父进程退出（exit）。这样做实现了下面几点：第一，如果该守护进程是作为一条简单shell命令启动的，那么父进程终止使得shell认为这条命令已经执行完毕；第二，子进程继承了父进程的进程组ID，但具有一个新的进程ID，这就保证了子进程不是一个进程组的组长进程。这对于下面就要做的setsid调用是必要的前提条件。

(3) 调用setsid以创建一个新会话。于是执行9.5节中列举的三个操作，使调用进程：(a) 成为新会话的首进程，(b) 成为一个新进程组的组长进程，(c) 没有控制终端。

425

> 在基于系统V的系统中，有些人建议在此时再次调用fork，并使父进程终止。第二个子进程作为守护进程继续运行。这样就保证了该守护进程不是会话首进程，于是按照系统V规则（见9.6节）可以防止它取得控制终端。避免取得控制终端的另一种方法是，无论何时打开一个终端设备都一定要指定O_NOCTTY。

(4) 将当前工作目录更改为根目录。从父进程处继承过来的当前工作目录可能在一个装配文件系统中。因为守护进程通常在系统再引导之前是一直存在的，所以如果守护进程的当前工作目录在一个装配文件系统中，那么该文件系统就不能被拆卸。这与装配文件系统的原意不符。

另外，某些守护进程可能会把当前工作目录更改到某个指定位置，在那里做它们的工作。例如，行式打印机假脱机守护进程常常将其工作目录更改到它们的spool目录上。

(5) 关闭不再需要的文件描述符。这使守护进程不再持有从其父进程继承来的某些文件描述符（父进程可能是shell进程，或某个其他进程）。可以使用程序清单2-4中的open_max函数或getrlimit函数（7.11节）来判定最高文件描述符值，并关闭直到该值的所有描述符。

(6) 某些守护进程打开/dev/null使其具有文件描述符0、1和2，这样，任何一个试图读标准输入、写标准输出或标准出错的库例程都不会产生任何效果。因为守护进程并不与终端设备相关联，所以不能在终端设备上显示其输出，也无处从交互式用户那里接收输入。即使守护进程是从交互式会话启动的，但因为守护进程是在后台运行的，所以登录会话的终止并不影响守护进程。如果其他用户在同一终端设备上登录，我们也不会在该终端上见到守护进程的输出，用户也不可期望他们在终端上的输入会由守护进程读取。

实例

程序清单13-1是个函数，可由想初始化成为一个守护进程的程序调用。

程序清单13-1 初始化一个守护进程

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int                 i, fd0, fd1, fd2;
    pid_t               pid;
    struct rlimit       rl;
    struct sigaction    sa;
```

426

```
    /*
     * Clear file creation mask.
     */
    umask(0);

    /*
     * Get maximum number of file descriptors.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: can't get file limit", cmd);

    /*
     * Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);
    setsid();

    /*
     * Ensure future opens won't allocate controlling TTYs.
     */
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't ignore SIGHUP");
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);

    /*
     * Change the current working directory to the root so
     * we won't prevent file systems from being unmounted.
     */
    if (chdir("/") < 0)
        err_quit("%s: can't change directory to /");

    /*
     * Close all open file descriptors.
     */
    if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024;
    for (i = 0; i < rl.rlim_max; i++)
        close(i);

    /*
     * Attach file descriptors 0, 1, and 2 to /dev/null.
     */
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     * Initialize the log file.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
```

427

```
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
          fd0, fd1, fd2);
        exit(1);
    }
}
```

若daemonize函数由main程序调用，然后main程序进入休眠状态，那么可以用ps命令检查该守护进程的状态：

```
$ ./a.out
$ ps -axj
 PPID    PID  PGID   SID TTY TPGID UID   COMMAND
    1   3346  3345  3345 ?      -1 501   ./a.out
$ ps -axj | grap 3345
    1   3346  3345  3345 ?      -1 501   ./a.out
```

也可用ps命令验证，没有一个活动进程的ID是3345。这意味着，我们的守护进程在一个孤儿进程组中（9.10节），它不是一个会话首进程，于是不会有机会分配到一个控制终端。这是在daemonize函数中执行第二个fork造成的。由此可以见到，此守护进程已经被正确地初始化了。□

13.4 出错记录

与守护进程有关的一个问题是如何处理出错消息。因为它没有控制终端，所以不能只是简单地写到标准出错上。在很多工作站上，控制台设备运行一个窗口系统，所以我们不希望所有守护进程都写到控制台设备上。我们也不希望每个守护进程将它自己的出错消息写到一个单独的文件中。对系统管理人员而言，如果要关心哪一个守护进程写到哪一个记录文件中，并定期地检查这些文件，那么一定会使他感到头痛。所以，需要有一个集中的守护进程出错记录设施。

> 在伯克利开发了BSD syslog设施，并广泛应用于4.2BSD。从BSD派生的很多系统都支持syslog。
>
> 在SVR4之前，系统V中从来没有一个集中的守护进程记录设施。
>
> 在Single UNIX Specification的XSI扩展中包括了syslog函数。

428

自4.2BSD以来，BSD syslog设施得到了广泛应用。大多数守护进程使用这一设施。图13-1显示了syslog设施的详细组织结构。

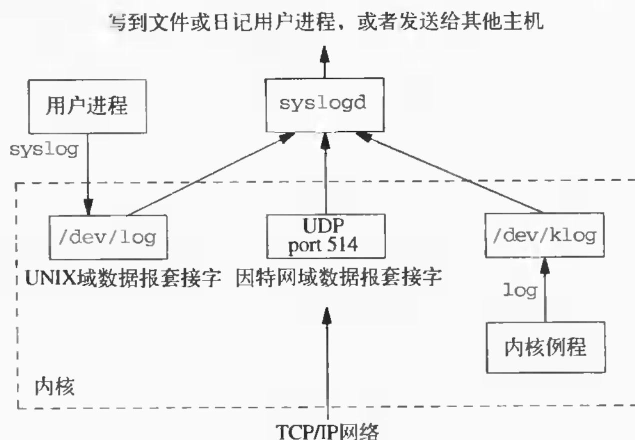


图13-1 BSD syslog设施

有三种方法产生日志消息：

(1) 内核例程可以调用log函数。任何一个用户进程通过打开（open）然后读（read）/dev/klog设备就可以读取这些消息。因为我们无意编写内核例程，所以不再进一步说明此函数。

(2) 大多数用户进程（守护进程）调用syslog(3)函数以产生日志消息。我们将在下面说明其调用序列。这使消息发送至UNIX域数据报套接字/dev/log。

(3) 在此主机上的一个用户进程，或通过TCP/IP网络连接到此主机的其他主机上的一个用户进程可将日志消息发向UDP端口514。注意，syslog函数并不产生这些UDP数据报，而是要求产生此日志消息的进程进行显式的网络编程。

关于UNIX域套接字以及UDP套接字的细节，请参阅Stevens, Fenner, and Rudoff [2004]。

通常，syslogd守护进程读取三种格式的日志消息。此守护进程在启动时读一个配置文件，一般其文件名为/etc/syslog.conf，该文件决定了不同种类的消息应送向何处。例如，紧急消息可被送向系统管理员（若已登录），并在控制台上显示，而警告消息则可记录到一个文件中。

429

该设施的接口是syslog函数。

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);

void syslog(int priority, const char *format, ...);

void closelog(void);

int setlogmask(int maskpri);

                                    返回值：前日志记录优先级屏蔽值
```

调用openlog是可选择的。如果不调用openlog，则在第一次调用syslog时，自动调用openlog。调用closelog也是可选择的——它只是关闭曾被用于与syslogd守护进程通信的描述符。

调用openlog使我们可以指定一个*ident*，将它加至每则日志消息中。*ident*一般是程序的名称（例如，cron、inetd等）。*option*参数是指定许多选项的位屏蔽。表13-1说明了可用的*option*（选项）。若某选项在Single UNIX Specification的openlog定义中已包括，则其XSI列用一个黑点表示。

表13-1　openlog的*option*参数

option	XSI	说　明
LOG_CONS	•	若日志消息不能通过UNIX域数据报送至syslogd，则将该消息写至控制台
LOG_NDELAY	•	立即打开至syslogd守护进程的UNIX域数据报套接字；不要等到记录了第一条消息。通常，在记录第一条消息之前，不打开该套接字
LOG_NOWAIT	•	不等待在将消息记入日志过程中可能已创建的子进程。因为在syslog调用wait时，应用程序可能已获得子进程的状态，这种处理阻止了与捕捉SIGCHLD信号的应用程序的冲突
LOG_ODELAY	•	在记录第一条消息之前延迟打开至syslogd守护进程的连接
LOG_PERROR		除将日志消息发送给syslogd外，还将它写至标准出错（在Solaris上不可用）
LOG_PID	•	每条消息都包含进程ID。此选项可供对每个请求都调用fork产生一个子进程的守护进程使用（与从不调用fork的守护进程比如syslogd对比）

openlog的参数*facility*可以选取表13-2中列举的值。注意，Single UNIX Specification只定义了*facility*参数值的一个子集，该子集是在一个给定的平台上典型地可用的。设置*facility*（设施）参数的目的是可以让配置文件说明，来自不同设施的消息将以不同的方式进行处理。如果不调用openlog，或者以*facility*为0来调用它，那么在调用*syslog*时，可将设施作为*priority*参数的一个部分进行说明。

430

表13-2　openlog的*facility*参数

facility	XSI	说　明
LOG_AUTH		授权程序：login，su，getty等
LOG_AUTHPRIV		与LOG_AUTH相同，但写日志文件时具有权限限制
LOG_CRON		cron和at
LOG_DAEMON		系统守护进程：inetd，routed等
LOG_FTP		FTP 守护进程（ftpd）
LOG_KERN		内核产生的消息
LOG_LOCAL0	•	保留由本地使用
LOG_LOCAL1	•	保留由本地使用
LOG_LOCAL2	•	保留由本地使用
LOG_LOCAL3	•	保留由本地使用
LOG_LOCAL4	•	保留由本地使用
LOG_LOCAL5	•	保留由本地使用
LOG_LOCAL6	•	保留由本地使用
LOG_LOCAL7	•	保留由本地使用
LOG_LPR		行打印系统：lpd，lpc等
LOG_MAIL		邮件系统
LOG_NEWS		Usenet网络新闻系统
LOG_SYSLOG		syslogd守护进程本身
LOG_USER	•	来自其他用户进程的消息（默认）
LOG_UUCP		UUCP系统

调用syslog产生一个日志消息。其*priority*参数是*facility*和*level*的组合，它们可选取的值分别列于表13-2和表13-3中。*level*值按优先级从最高到最低按序排列。

表13-3　syslog中的*level*（按序排列）

level	说　明
LOG_EMERG	紧急状态（系统不可使用）（最高优先级）
LOG_ALERT	必须立即修复的状态
LOG_CRIT	严重状态（例如，硬设备出错）
LOG_ERR	出错状态
LOG_WARNING	警告状态
LOG_NOTICE	正常，但重要的状态
LOG_INFO	信息性消息
LOG_DEBUG	调试消息（最低优先级）

*format*参数以及其他参数传至vsprintf函数以便进行格式化。在*format*中，每个%m都先被代换成对应于errno值的出错消息字符串（strerror）。

setlogmask函数用于设置进程的记录优先级屏蔽字。它返回调用它之前的屏蔽字。当设

置了记录优先级屏蔽字时，除非消息的优先级已在记录优先级屏蔽字中设置，否则消息不被记录。注意，试图将该屏蔽字设置为0并不产生任何作用。

431

很多系统也提供logger(1)程序，以其作为向syslog设施发送日志消息的方法。虽然Single UNIX Specification并未定义任何可选参数，但某些实现还是允许该程序的可选参数指定*facility*、*level*以及*ident*。logger命令本是为了用于以非交互方式运行但又要产生日志消息的shell脚本的。

实　例

在一个（假定的）行式打印机假脱机守护进程中，可能包含有下面的调用序列：

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

第一个调用将*ident*字符串设置为程序名，指定打印该进程ID，并且将系统默认的*facility*设定为行式打印机系统。对syslog的调用指定一个出错状态和一个消息字符串。如若不调用openlog，则第二个调用的形式可能是：

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

其中，将*priority*参数指定为*level*和*facility*的组合。□

除了syslog，很多平台还提供它的一种变体处理可变参数列表。

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

本书说明的所有四种平台都提供vsyslog，但Single UNIX Specification并不包括它。

大多数syslog实现将使消息短时间处于队列中。如果在此段时间中到达了重复消息，那么syslog守护进程将不把它写到日记记录中，而是打印输出一条消息，类似于“上一条消息重复了*N*次”。

13.5　单实例守护进程

为了正常运作，某些守护进程实现为单实例的，也就是在任一时刻只运行该守护进程的一个副本。例如，该守护进程可能需要排它地访问一个设备。在cron守护进程情况下，如果同时有多个实例运行，那么每个副本都可能试图开始某个预定的操作，于是造成该操作的重复执行，这很可能导致出错。

如果守护进程需要访问一设备，而该设备驱动程序将阻止多次打开在/dev目录下的相应设备节点，那么这就达到了任何时刻只运行守护进程一个副本的要求。但是如果没有这种设备可供使用，那么我们就需要自行处理。

432

文件锁和记录锁机制是一种方法的基础，该方法用来保证一个守护进程只有一个副本在运行。（在14.3节中再来讨论文件和记录锁。）如果每一个守护进程创建一个文件，并且在整个文件上加上一把写锁，那就只允许创建一把这样的写锁，所以在此之后如试图再创建一把这样的写锁就将失败，以此向后续守护进程副本指明已有一个副本正在运行。

文件和记录锁提供了一种方便的互斥机制。如果守护进程在整个文件上得到一把写锁，那么在该守护进程终止时，这把锁将被自动删除。这就简化了复原所需的处理，去除了对以前的守护进程实例需要进行清理的有关操作。

实例

程序清单13-2中的函数说明了如何使用文件和记录锁以保证只运行某守护进程的一个副本。

程序清单13-2 保证只运行某个守护进程的一个副本

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);

int
already_running(void)
{
    int     fd;
    char    buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

433

守护进程的每个副本都将试图创建一个文件，并将其进程ID写到该文件中。这使管理人员易于标识该进程。如果该文件已经加了锁，那么lockfile函数将失败，errno设置为EACCES或EAGAIN，函数返回1，这表明该守护进程已在运行。否则将文件长度截短为0，将进程ID写入该文件，函数返回0。

我们需要将文件长度截短为0，其原因是以前守护进程实例的进程ID字符串可能长于调用此函数的当前进程的进程ID字符串。例如，若以前的守护进程的进程ID是12345，而新实例的进程ID是9999，那么将此进程ID写入文件后，在文件中留下的是99995。将文件长度截短为0就

解决了此问题。 □

13.6 守护进程的惯例

在UNIX系统中，守护进程遵循下列公共惯例：

- 若守护进程使用锁文件，那么该文件通常存放在/var/run目录中。注意，守护进程可能需具有超级用户权限才能在此目录下创建文件。锁文件的名字通常是*name*.pid，其中，*name*是该守护进程或服务的名字。例如，cron守护进程锁文件的名字是/var/run/crond.pid。
- 若守护进程支持配置选项，那么配置文件通常存放在/etc目录中。配置文件的名字通常是*name*.conf，其中，*name*是该守护进程或服务的名字。例如，syslogd守护进程的配置文件是/etc/syslog.conf。
- 守护进程可用命令行启动，但通常它们是由系统初始化脚本之一（/etc/rc*或/etc/init.d/*）启动的。如果在守护进程终止时，应当自动地重新启动它，则我们可在/etc/inittab中为该守护进程包括_respawn记录项，这样，init就将重启动该守护进程。
- 若一守护进程有一配置文件，那么当该守护进程启动时，它读该文件，但在此之后一般就不会再查看它。若一管理员更改了配置文件，那么该守护进程可能需要被停止，然后再启动，以使配置文件的更改生效。为避免此种麻烦，某些守护进程将捕捉SIGHUP信号，当它们接收到该信号时，重读配置文件。因为守护进程并不与终端相结合，它们或者是无控制终端的会话首进程，或者是孤儿进程组的成员，所以守护进程并不期望接收SIGHUP。于是，它们可以安全地重复使用它。

434

程序清单13-3所示程序说明了守护进程可以重读其配置文件的一种方法。该程序使用sigwait以及多线程，对此我们已经在12.8节讨论过。

程序清单13-3　守护进程重读配置文件

```
#include "apue.h"
#include <pthread.h>
#include <syslog.h>

sigset_t    mask;

extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;
```

```
    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "sigwait failed");
            exit(1);
        }

        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Re-reading configuration file");
            reread();
            break;

        case SIGTERM:
            syslog(LOG_INFO, "got SIGTERM; exiting");
            exit(0);

        default:
            syslog(LOG_INFO, "unexpected signal %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int                 err;
    pthread_t           tid;
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Restore SIGHUP default and block all signals.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't restore SIGHUP default");
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "SIG_BLOCK error");
```

435

```
    /*
     * Create a thread to handle SIGHUP and SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

436

该程序调用程序清单13-1中的daemonize以初始化守护进程。从该函数返回后，调用程序清单13-2中的already_running函数以确保该守护进程只有一个副本在运行。到达这一点时，SIGHUP信号仍被忽略，所以需恢复对该信号的系统默认处理方式；否则调用sigwait的线程决不会见到该信号。

如同对多线程程序所推荐的那样，我们阻塞所有信号，然后创建一线程，由它来处理信号。该线程的唯一工作是等待SIGHUP和SIGTERM。当接收到SIGHUP信号时，该线程调用reread函数重读它的配置文件。当它接收到SIGTERM信号时，记录一消息，然后终止。

回忆表10-1，对于SIGHUP和SIGTERM的默认动作是终止进程。因为我们阻塞了这些信号，所以当对进程产生这些信号时，守护进程不会消亡，而是调用sigwait的线程在返回时将指示已接收到该信号。 □

如在12.8节所说明的那样，Linux线程对于信号的处理方式与众不同。由于这一点，在程序清单13-3中对信号标识合适的进程是困难的。另外，由于实现的差别，不能保证守护进程将按所期望的那样做出反应。

程序清单13-4说明守护进程无需使用多线程也可以捕捉SIGHUP并重读其配置文件。

程序清单13-4 守护进程重读配置文件的另一种实现

```
#include "apue.h"
#include <syslog.h>
#include <errno.h>

extern int lockfile(int);
extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void
sigterm(int signo)
{
    syslog(LOG_INFO, "got SIGTERM; exiting");
    exit(0);
}
```

```
void
sighup(int signo)
{
    syslog(LOG_INFO, "Re-reading configuration file");
    reread();
}
int
main(int argc, char *argv[])
{
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Handle signals of interest.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGTERM: %s", strerror(errno));
        exit(1);
    }
    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGHUP: %s", strerror(errno));
        exit(1);
    }

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

437

438

在初始化守护进程后，我们为SIGHUP和SIGTERM配置信号处理程序。我们可以将重读逻辑放在信号处理程序中，也可以只在其中设置一个标志，由守护进程的主线程做所有所需的工作。 □

13.7 客户进程-服务器进程模型

守护进程常常用作服务器进程。确实，我们可以称图13-1中的syslogd进程为服务器进程，用户进程（客户进程）用UNIX域数据报套接字向其发送消息。

一般而言，*服务器*是等待*客户*进程与其联系的一个进程，客户进程向它提出某种类型的服务要求。图13-1中，由syslogd服务器进程提供的服务是将出错消息记录到日志文件中。

图13-1中，客户进程和服务器进程之间的通信是单向的。客户进程向服务器进程发送服务请求，服务器进程则不向客户进程回送任何消息。在接下来有关进程通信的几章中，我们将见到大量客户进程和服务器进程之间双向通信的实例。客户进程向服务器进程发送请求，服务器进程则向客户进程回送应答。

13.8 小结

在大多数UNIX系统中，守护进程是一直运行的。为了初始化我们自己的守护进程，需要审慎思索并深入理解第9章中说明过的进程之间的关系。本章开发了一个可由守护进程调用的、对其自身正确地进行初始化的函数。

因为守护进程通常没有控制终端，所以本章还讨论了守护进程记录出错消息的几种方法。我们讨论了在大多数UNIX系统中，守护进程遵循的若干惯例，给出了几个如何实现某些惯例的实例。

习题

13.1 从图13-1可以看出，直接调用openlog或第一次调用syslog都可以初始化syslog设施，此时一定要打开用于UNIX域的数据报套接字的特殊设备文件/dev/log。如果调用openlog前，用户进程（守护进程）先调用了chroot，结果如何？

13.2 列出你的系统中所有活动的守护进程，并说明它们的功能。

13.3 编写一段调用程序清单13-1中daemonize函数的程序。调用该函数后再调用getlogin（见8.15节）查看该进程是否有登录名（既然它现在已成为守护进程）。将结果打印到一个文件中。

439

第14章

高级I/O

14.1 引言

本章内容包括非阻塞I/O、记录锁、系统V流机制、I/O多路转接（select和poll函数）、readv和writev函数以及存储映射I/O（mmap），这些都称为高级I/O。第15章、第17章中的进程间通信，以及以后各章中的很多实例都要使用本章所描述的概念和函数。

14.2 非阻塞I/O

10.5节中曾将系统调用分成“低速”系统调用和其他系统调用两类。低速系统调用是可能会使进程永远阻塞的一类系统调用，它们包括下列调用：

- 如果某些文件类型（例如管道、终端设备和网络设备）的数据并不存在，则读操作可能会使调用者永远阻塞。
- 如果数据不能立即被上述同样类型的文件接受（由于在管道中无空间、网络流控制等），则写操作也会使调用者永远阻塞。
- 在某种条件发生之前，打开某些类型的文件会被阻塞（例如打开一个终端设备可能需等到与之连接的调制解调器应答；又例如在没有其他进程已用读模式打开该FIFO时若以只写模式打开FIFO，那么也要等待）。
- 对已经加上强制性记录锁的文件进行读、写。
- 某些ioctl操作。
- 某些进程间通信函数（见第15章）。

我们也曾说过，虽然读、写磁盘文件会使调用者在短暂时间内阻塞，但并不能将与磁盘I/O有关的系统调用视为“低速”。

非阻塞I/O使我们可以调用open、read和write这样的I/O操作，并使这些操作不会永远阻塞。如果这种操作不能完成，则调用立即出错返回，表示该操作如继续执行将阻塞。

对于一个给定的描述符有两种方法对其指定非阻塞I/O：

(1) 如果调用open获得描述符，则可指定O_NONBLOCK标志（见3.3节）。

(2) 对于已经打开的一个描述符，则可调用fcntl，由该函数打开O_NONBLOCK文件状态标志（见3.14节）。程序清单3-5中的函数可用来为一个描述符打开任一文件状态标志。

系统V的早期版本使用标志O_NDELAY指定非阻塞方式。在这些版本中，如果无数据可读，则read返回值0。而UNIX系统又常将read的返回值0解释为文件结束，两者有所混淆。因此POSIX.1提供了一

个名字和语义都与O_NDELAY不同的非阻塞标志。确实，在系统V的早期版本中，当从read得到返回值0时，我们并不知道该调用是否阻塞了或已到文件结尾处。POSIX.1要求，对于一个非阻塞的描述符如果无数据可读，则read返回-1，并且errno被设置为EAGAIN。系统V派生的某些平台支持较老的O_NDELAY和POSIX.1的O_NONBLOCK两者，但在本书的实例中只使用POSIX.1的规定。O_NDELAY只是为了向后兼容，不应在新应用程序中使用。

4.3BSD为fcntl提供了FNDELAY标志，其语义也稍有区别。它不只影响描述符的文件状态标志，还将终端设备或套接字的标志更改成非阻塞的，因此不仅影响共享同一文件表项的用户，而且对终端或套接字的所有用户起作用（4.3BSD非阻塞I/O只对终端和套接字起作用）。另外，如果对一个非阻塞描述符的操作不能无阻塞地完成，那么4.3BSD返回EWOULDBLOCK。现今，基于BSD的系统提供POSIX.1的O_NONBLOCK标志，并且将EWOULDBLOCK定义为与POSIX.1的EAGAIN相同。这些系统提供与其他依从POSIX系统相一致的非阻塞语义。文件状态标志的更改影响同一文件表项的所有用户，但与通过其他文件表项对同一设备的访问无关（参见图3-1和图3-3）。

实例

程序清单14-1是一个非阻塞I/O的实例，它从标准输入读500 000字节，并试图将它们写到标准输出上。该程序先将标准输出设置为非阻塞的，然后用for循环进行输出，每次write调用的结果都在标准出错上打印。函数clr-fl类似于程序清单3-5中的set_fl，但与set_fl的功能相反，它清除1个或多个标志位。

442

程序清单14-1 长的非阻塞write

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>

char    buf[500000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }
```

```
    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */

    exit(0);
}
```

若标准输出是普通文件，则可以期望write只执行一次：

```
$ ls -l /etc/termcap                                    打印文件长度
-rw-r--r--  1 root     702559 Feb 23  2002 /etc/termcap
$ ./a.out < /etc/termcap > temp.file                    先试一普通文件
read 500000 bytes
nwrite = 500000, errno = 0                              一次写
$ ls -l temp.file                                       检验输出文件长度
-rw-rw-r--  1 sar      500000 Jul  8 04:19 temp.file
```

但是，若标准输出是终端，则期望write有时会返回小于500 000的一个数字，有时则出错返回。下面是在一个系统上运行上述程序的结果：

443

```
$ ./a.out < /etc/termcap 2>stderr.out                   输出至终端
                                                        大量输出至终端……
$ cat stderr.out
read 500000 bytes
nwrite = 216041, errno = 0
nwrite = -1, errno = 11                                 这种错1 497次……
. . .
nwrite = 16015, errno = 0
nwrite = -1, errno = 11                                 这种错1 856次……
. . .
nwrite = 32081, errno = 0
nwrite = -1, errno = 11                                 这种错1 654次……
. . .
nwrite = 48002, errno = 0
nwrite = -1, errno = 11                                 这种错1 460次……
. . .
                                                        等等
nwrite = 7949, errno = 0
```

在该系统上，errno值11对应的是EAGAIN。终端驱动程序一次接收的数据量随系统而变。根据你登录系统时所使用的不同方式——是在系统控制台上登录，还是在硬接线的终端上登录，或是用伪终端在网络连接上登录——该程序运行的结果也不同。如果你在终端上在运行一窗口系统，那么也是经由伪终端设备与系统交互。 □

在此实例中，程序发出了数千个write调用，但是只有10~20个左右是真正输出数据的，其余的则出错返回。这种形式的循环称为轮询，在多用户系统上它浪费了CPU时间。14.5节将介绍非阻塞描述符的I/O多路转接，这是进行这种操作的一种比较有效的方法。

有时，我们可以将应用程序设计成使用多线程（见第11章），从而避免使用非阻塞I/O。如若我们能在其他线程中继续进展，则可以允许某个线程在I/O调用中阻塞。这种方法有时能简化应用程序的设计（见第21章），但是线程间同步的开销有时却可能增加复杂性，于是导致得不偿失的后果。

14.3 记录锁

若两个人同时编辑一个文件，其后果将如何呢？在很多UNIX系统中，该文件的最后状态

取决于写该文件的最后一个进程。但是对于有些应用程序（例如数据库），进程有时需要确保它正在单独写一个文件。为了向进程提供这种功能，商用UNIX系统提供了记录锁机制。（第20章我们开发了一个使用记录锁的数据库函数库。）

记录锁（record locking）的功能是：当一个进程正在读或修改文件的某个部分时，它可以阻止其他进程修改同一文件区。对于UNIX系统而言，“记录”这个词是一种误用，因为UNIX系统内核根本没有使用文件记录这种概念。更适合的术语可能是**字节范围锁**（byte-range locking），因为它锁定的只是文件中的一个区域（也可能是整个文件）。

444

1. 历史

对早期UNIX系统的一种批评是它们不能用来运行数据库系统，其原因是这些系统不支持部分地对文件加锁。在UNIX系统开始进入商用计算领域时，很多系统开发小组以各种不同方式增加了对记录锁的支持。

早期的伯克利版本只支持`flock`函数。该函数锁整个文件，不能锁文件中的一部分。

SVR3通过`fcntl`函数增加了记录锁功能。在此基础上构造了`lockf`函数，它提供了一个简化的接口。这些函数允许调用者锁一个文件中任意字节数的区域，长至整个文件，短至文件中的一个字节。

POSIX.1标准的基础是`fcntl`。表14-1列出了各种UNIX系统提供的不同形式的记录锁。注意，Single UNIX Specification在其XSI扩展中包括了`lockf`。

表14-1　各种UNIX系统支持的记录锁形式

系　统	建议性	强制性	fcntl	lockf	flock
SUS	•		•	XSI	
FreeBSD 5.2.1	•		•	•	•
Linux 2.4.22	•	•	•	•	•
Mac OS X 10.3	•		•	•	•
Solaris 9	•	•	•	•	•

本节最后部分将说明建议性锁和强制性锁之间的区别。本书只介绍POSIX.1的`fcntl`锁。

> 记录锁是1980年由John Bass最早加到V7上的。内核中相应的系统调用入口项是名为`locking`的函数。此函数提供了强制性记录锁功能，它被用在很多System III版本中。Xenix系统采用了此函数，某些基于Intel的系统V派生版本（例如OpenServer 5），在Xenix兼容库中仍旧支持该函数。

2. `fcntl`记录锁

3.14节中已经给出了`fcntl`函数的原型，为了叙述方便，这里再重复一次。

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );
```

返回值：若成功则依赖于*cmd*（见下），若出错则返回−1

445

对于记录锁，*cmd*是`F_GETLK`、`F_SETLK`或`F_SETLKW`。第三个参数（称其为*flockptr*）是一个指向`flock`结构的指针：

```
struct flock {
  short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
  off_t  l_start;  /* offset in bytes, relative to l_whence */
```

```
  short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;    /* returned with F_GETLK */
};
```

对flock结构说明如下：

- 所希望的锁类型：F_RDLCK（共享读锁）、F_WRLCK（独占性写锁）或F_UNLCK（解锁一个区域）。
- 要加锁或解锁区域的起始字节偏移量，这由l_start和l_whence两者决定。
- 区域的字节长度，由l_len表示。
- 具有能阻塞当前进程的锁，其持有进程的ID存放在l_pid中（仅由F_GETLK返回）。

关于加锁和解锁区域的说明还要注意下列各点：

- l_stat是相对偏移量（字节），l_whence则决定了相对偏移量的起点。这与lseek函数（见3.6节）中最后两个参数类似。确实，l_whence可选用的值是SEEK_SET、SEEK_CUR或SEEK_END。
- 该区域可以在当前文件尾端处开始或越过其尾端处开始，但是不能在文件起始位置之前开始。
- 如若l_len为0，则表示锁的区域从其起点（由l_start和l_whence决定）开始直至最大可能偏移量为止，也就是不管添写到该文件中多少数据，它们都处于锁的范围内（不必猜测会有多少字节被追加到文件之后）。
- 为了锁整个文件，我们设置l_start和l_whence，使锁的起点在文件起始处，并且说明长度（l_len）为0。（有多种方法可以指定文件起始处，但常用的方法是将l_start指定为0，l_whence指定为SEEK_SET。）

上面提到了两种类型的锁：共享读锁（l_type为F_RDLCK）和独占写锁（F_WRLCK）。基本规则是：多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定字节上只能有一个进程独用的一把写锁。进一步而言，如果在一个给定字节上已经有一把或多把读锁，则不能在该字节上再加写锁；如果在一个字节上已经有一把独占性的写锁，则不能再对它加任何读锁。在表14-2示出了这些规则。

446

表14-2 不同类型锁之间的兼容性

		请求	
		读锁	写锁
当前区域状态	无锁	允许	允许
	一个或多个读锁	允许	拒绝
	一个写锁	拒绝	拒绝

上面说明的兼容性规则适用于不同进程提出的锁请求，并不适用于单个进程提出的多个锁请求。如果一个进程对一个文件区间已经有了一把锁，后来该进程又企图在同一文件区间再加一把锁，那么新锁将替换老锁。例如，若一进程在某文件的16~32字节区间有一把写锁，然后又试图在16~32字节区间加一把读锁，那么该请求将成功执行（假定其他进程此时并不试图向该文件的同一区间加锁），原来的写锁被替换为读锁。

加读锁时，该描述符必须是读打开；加写锁时，该描述符必须是写打开。

以下说明fcntl函数的三种命令：

F_GETLK　判断由flockptr所描述的锁是否会被另外一把锁所排斥（阻塞）。如果存在一把锁，它阻止创建由flockptr所描述的锁，则把该现存锁的信息写到flockptr指向的结构中。如果不存在这种情况，则除了将l_type设置为F_UNLCK之外，flockptr所指向结构中的其他信息保持不变。

F_SETLK　设置由flockptr所描述的锁。如果试图建立一把读锁（l_type设为F_RDLCK）或写锁（l_type设为F_WRLCK），而按上述兼容性规则不能允许，则fcntl立即出错返回，此时errno设置为EACCES或EAGAIN。

> 虽然POSIX允许实现返回这两种出错代码中的任何一种，但本书说明的四种实现在锁请求不能得到满足时，都返回EAGAIN。

此命令也用来清除由flockptr说明的锁（l_type为F_UNLCK）。

F_SETLKW　这是F_SETLK的阻塞版本（命令名中的W表示等待（wait））。如果因为当前在所请求区间的某个部分另一个进程已经有一把锁，因而按兼容性规则由flockptr所请求的锁不能被创建，则使调用进程休眠。如果请求创建的锁已经可用，或者休眠由信号中断，则该进程被唤醒。

447

应当了解，用F_GETLK测试能否建立一把锁，然后用F_SETLK和F_SETLKW企图建立一把锁，这两者不是一个原子操作。因此不能保证在这两次fcntl调用之间不会有另一个进程插入并建立一把相关的锁，从而使原来测试到的情况发生变化。如果不希望在建立锁时可能产生的长期阻塞，则应使用F_SETLK，并对返回结果进行测试，以判别是否成功地建立了所要求的锁。

> 注意，POSIX.1并没有说明在下列情况下将发生什么：一个进程在某个文件的一个区间上设置了一把读锁，第二个进程试图对同一文件区间加一把写锁时阻塞，然后第三个进程则试图在同一文件区间上得到另一把读锁。如果第三个进程只是因为读区间已有一把读锁，而被允许在该区间放置另一把读锁，那么这种实现就可能会使希望加写锁的进程饿死。这意味着，当对同一区间加另一把读锁的请求到达时，提出加写锁而阻塞的进程需等待的时间延长了。如果加读锁的请求来得很频繁，使得该文件区间始终存在一把或几把读锁，那么欲加写锁的进程就将等待很长时间。

在设置或释放文件上的锁时，系统按要求组合或裂开相邻区。例如，若字节100~199是加锁的区，需解锁第150字节，则内核将维持两把锁，一把用于字节100~149，另一把用于字节151~199。图14-1说明了这种情况。

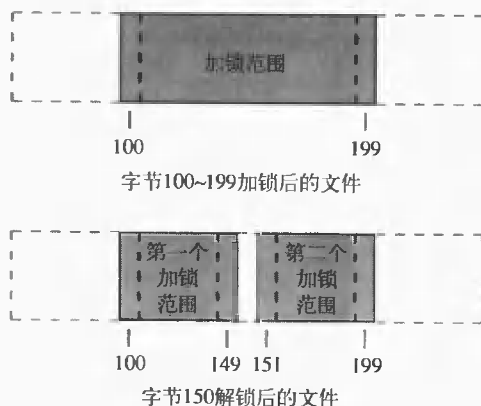


图14-1　文件字节范围锁

假定我们又对第150字节设置锁，那么系统将会把三个相邻的加锁区合并成一个区（从字节100至199）。其结果如图14-1中的第一图所示，于是我们又回到了出发点。

实例：请求和释放一把锁

为了避免每次分配flock结构，然后又填入各项信息，可以用程序清单14-2中的函数lock_reg来处理所有这些细节。

448

程序清单14-2 加锁和解锁一个文件区域的函数

```
#include "apue.h"
#include <fcntl.h>

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    return(fcntl(fd, cmd, &lock));
}
```

因为大多数锁调用是加锁或解锁一个文件区域（命令F_GETLK很少使用），故通常使用下列5个宏，它们都定义在apue.h中（见附录B）。

```
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

我们有目的地用与lseek函数同样的顺序定义这些宏中的前三个参数。 □

实例：测试一把锁

程序清单14-3中定义了一个函数lock_test，可用其测试一把锁。

程序清单14-3 测试一个锁状态的函数

449

```
#include "apue.h"
#include <fcntl.h>

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;
    lock.l_type = type;     /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
```

```
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);      /* false, region isn't locked by another proc */
    return(lock.l_pid); /* true, return pid of lock owner */
}
```

如果存在一把锁，它阻塞由参数说明的锁请求，则此函数返回持有这把现存锁的进程ID，否则此函数返回0。通常用下面两个宏来调用此函数（它们也定义在apue.h中）。

```
#define is_read_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

注意，进程不能使用lock_test函数测试它自己是否在文件的某一部分持有一把锁。F_GETLK命令的定义说明，返回信息指示是否有现存的锁阻止调用进程设置它自己的锁。因为F_SETLK和F_SETLKW命令总是替换调用进程现存的锁（若已存在），所以调用进程决不会阻塞在自己持有的锁上；于是，F_GETLK命令决不会报告调用进程自己持有的锁。 □

实例：死锁

如果两个进程相互等待对方持有并且锁定的资源时，则这两个进程就处于死锁状态。如果一个进程已经控制了文件中的一个加锁区域，然后它又试图对另一个进程控制的区域加锁，则它就会休眠，在这种情况下，有发生死锁的可能性。

程序清单14-4给出了一个死锁的例子。子进程锁字节0，父进程锁字节1。然后，它们又都试图锁对方已经加锁的字节。在该程序中使用了8.9节中介绍的父、子进程同步例程（TELL_xxx和WAIT_xxx），使得每个进程能够等待另一个进程获得它设置的第一把锁。运行程序清单14-4所示程序得到：

```
$ ./a.out
parent: got the lock, byte 1
child: got the lock, byte 0
child: writew_lock error: Resource deadlock avoided
parent: got the lock, byte 0
```

450

程序清单14-4　死锁检测实例

```
#include "apue.h"
#include <fcntl.h>

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);
    printf("%s: got the lock, byte %ld\n", name, offset);
}

int
main(void)
```

```
{
    int     fd;
    pid_t   pid;

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");

    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {                        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

检测到死锁时，内核必须选择一个进程接收出错返回。在本实例中选择了子进程，这是一个实现细节。在某些系统上，总是子进程接到出错信息；在另一些系统上，总是父进程接到出错信息。在某些系统上，当试图使用多把锁时，有时是子进程接到出错信息，有时则是父进程接到出错信息。 □ 451

3. 锁的隐含继承和释放

关于记录锁的自动继承和释放有三条规则：

(1) 锁与进程和文件两方面有关。这有两重含义：第一重很明显，当一个进程终止时，它所建立的锁全部释放；第二重意思就不很明显，任何时候关闭一个描述符时，则该进程通过这一描述符可以引用的文件上的任何一把锁都被释放（这些锁都是该进程设置的）。这就意味着如果执行下列四步：

```
fd1 = open(pathname, ...);
read_lock(fd1, ...);
fd2 = dup(fd1);
close(fd2);
```

则在close(fd2)后，在fd1上设置的锁被释放。如果将dup换为open，以打开另一描述符上的同一文件，其效果也一样：

```
fd1 = open(pathname, ...);
read_lock(fd1, ...);
fd2 = open(pathname, ...)
close(fd2);
```

(2) 由fork产生的子进程不继承父进程所设置的锁。这意味着，若一个进程得到一把锁，然后调用fork，那么对于父进程获得的锁而言，子进程被视为另一个进程，对于从父进程处

继承过来的任一描述符，子进程需要调用fcntl才能获得它自己的锁。这与锁的作用是相一致的。锁的作用是阻止多个进程同时写同一个文件（或同一文件区域）。如果子进程继承父进程的锁，则父、子进程就可以同时写同一个文件。

(3) 在执行exec后，新程序可以继承原执行程序的锁。但是注意，如果对一个文件描述符设置了close-on-exec标志，那么当作为exec的一部分关闭该文件描述符时，对相应文件的所有锁都被释放了。

4. FreeBSD的实现

先简要地观察FreeBSD实现中使用的数据结构。这会帮助我们进一步理解规则1：锁是与进程、文件两者相关联的。

考虑一个进程，它执行下列语句（忽略出错返回）：

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);    /* parent write locks byte 0 */
if ((pid = fork()) > 0) {           /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
}
pause();
```

452

图14-2显示了父、子进程暂停后的数据结构情况。

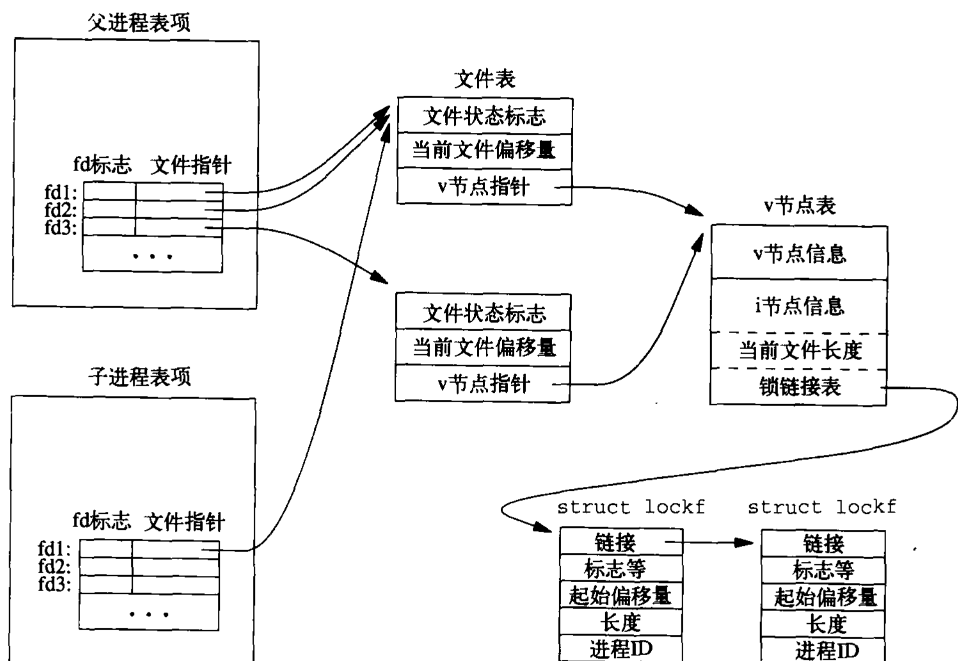


图14-2　关于记录锁的FreeBSD数据结构

图3-3和图8-1中已显示了open、fork以及dup后的数据结构。有了记录锁后，在原来的这些图上新加了lockf结构，它们由i节点结构开始相互链接起来。注意，每个lockf结构说明了一个给定进程的一个加锁区域（由偏移量和长度定义）。图中显示了两个lockf结构，一个是由父进程调用write_lock形成的，另一个则是由子进程调用read_lock形成的。每一个结

构都包含了相应进程ID。

在父进程中，关闭fd1、fd2和fd3中的任意一个都将释放由父进程设置的写锁。在关闭这三个描述符中的任意一个时，内核会从该描述符所关联的i节点开始，逐个检查lockf链接表中各项，并释放由调用进程持有的各把锁。内核并不清楚也不关心父进程是用哪一个描述符来设置这把锁的。

在程序清单13-2中，我们了解到，守护进程可用一把文件锁以保证只有该守护进程的唯一副本正在运行。程序清单14-5示出了lockfile函数的实现，守护进程可用该函数在文件上加锁。　453

程序清单14-5　在文件整体上加锁

```
#include <unistd.h>
#include <fcntl.h>

int
lockfile(int fd)
{
    struct flock fl;

    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

另一种方法是，用write_lock函数定义lockfile函数：

```
#define lockfile(fd) write_lock((fd), 0, SEEK_SET, 0)
```

□

5. 在文件尾端加锁

在接近文件尾端加锁或解锁时需要特别小心。大多数实现按照l_whence的SEEK_CUR或SEEK_END值，用l_start以及文件当前位置或当前长度得到绝对文件偏移量。但是，常常需要相对于文件的当前位置或当前长度指定一把锁。其原因是，我们在该文件上没有锁，所以不能调用lseek以正确无误地获得加锁时的当前文件偏移量。（在lseek和加锁调用之间，另一个进程可能改变该文件长度。）

考虑以下代码序列：

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

该代码序列所做的可能并不是你所期望的。它得到一把写锁，该写锁从当前文件尾端起，包括以后可能添加到该文件的任何数据。假定在文件尾端时执行第一个write，它给文件添写了1个字节，而该字节将被加锁。跟随其后的解锁，其作用是对以后添写到文件上的数据不再加锁，但在它之前刚添写的一个字节则保留加锁。当执行第二个写时，文件尾端又延伸了1个字节，但该字节并未加锁。由此代码序列造成的文件锁状态示于图14-3。　454

当对文件的一部分加锁时，内核将指定的偏移量变换成绝对文件偏移量。另外，除了指定一个绝对偏移量（SEEK_SET）之外，fcntl还允许我们相对于文件中的某个点（当前偏移量

（SEEK_CUR）或文件尾端（SEEK_END））指定该偏移量。当前偏移量和文件尾端是可能不断变化的，而这种变化又不应影响现存锁的状态，所以内核必须独立于当前文件偏移量或文件尾端而记住锁。

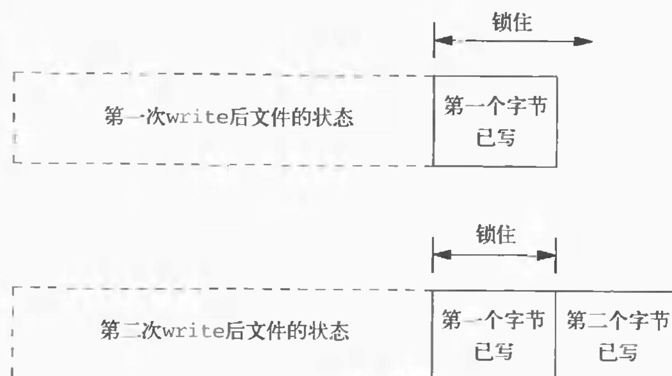


图14-3　文件范围锁

如果我们想解除第一次write所写1个字节上的锁，那么应指定长度为−1。负的长度值表示在指定偏移量之前的字节数。

6. 建议性锁和强制性锁

考虑数据库访问例程库。如果该库中所有函数都以一致的方法处理记录锁，则称使用这些函数访问数据库的任何进程集为*合作进程*（cooperating process）。如果这些函数是仅有地用来访问数据库的函数，那么它们使用建议性锁是可行的。但是建议性锁并不能阻止对数据库文件有写权限的任何其他进程对数据库文件进行随意的写操作。没有使用被认可的方法（数据库函数库）访问数据库的进程是一个非合作进程。

强制性锁使内核对每一个open、read和write系统调用都进行检查，检查调用进程对正在访问的文件是否违背了某一把锁的作用。强制性锁有时也被称为*强迫方式锁*（enforcement-mode locking）。

> 从表14-1看到，Linux 2.4.22和Solaris 9提供强制性记录锁，而FreeBSD 5.2.1和Mac OS X 10.3则不提供。强制性记录锁不是Single UNIX Specification的组成部分。在Linux中，如果用户想要使用强制性锁，则要在各个文件系统基础上、对mount命令用_omand选项打开该机制。

455

对一个特定文件打开其设置组ID位并关闭其组执行位，则对该文件开启了强制性锁机制（回忆程序清单4-4）。因为当组执行位关闭时，设置组ID位不再有意义，所以SVR3的设计者借用两者的这种组合来指定对一个文件的锁是强制性的而非建议性的。

如果一个进程试图读、写一个强制性锁起作用的文件，而欲读、写的部分又由其他进程加上了读或写锁，此时会发生什么呢？对这一问题的回答取决于三方面的因素：操作类型（read或write），其他进程保有的锁的类型（读锁或写锁），以及有关描述符是阻塞还是非阻塞的。表14-3列出了8种可能性。

除了表14-3中的read和write函数，其他进程持有的强制性锁也会对open函数产生影响。通常，即使正在打开的文件具有强制性记录锁，该打开操作也会成功。后随的read或write依从于表14-3中所示的规则。但是，如果欲打开的文件具有强制性记录锁（读锁或写锁），而且open调用中的flag指定为O_TRUNC或O_CREAT，则不论是否指定O_NONBLOCK，open都

立即出错返回，errno设置为EAGAIN。

表14-3　强制性锁对其他进程读、写的影响

其他进程在文件区段中持有的现存锁的类型	阻塞描述符，试图		非阻塞描述符，试图	
	read	write	read	write
读锁	允许	阻塞	允许	EAGAIN
写锁	阻塞	阻塞	EAGAIN	EAGAIN

只有Solaris对O_CREAT标志处理为出错。当打开一个具强制性锁的文件时，Linux允许指定O_CREAT标志。对O_TRUNC标志产生open出错是有道理的，因为若其他进程对该文件持有读、写锁，那么就不能将其截短为0。对O_CREAT标志在返回时也设置errno则无道理，因为该标志的意义是如果该文件不存在则创建，由于其他进程对该文件持有记录锁，因而该文件肯定是存在的。

这种open的锁冲突处理方式可能导致令人惊异的结果。我们曾编写过一个测试程序，它打开一个文件（其模式指定为强制性锁），然后对该文件的整体设置一把读锁，然后进入休眠一段时间。（回忆表14-3，读锁应当阻止其他进程写该文件。）在这段休眠时间内，用某些典型的UNIX系统程序和操作符对该文件进行处理，发现下列情况：

- 可用ed编辑程序对该文件进行编辑操作，而且编辑结果可以写回磁盘！强制性记录锁对此毫无影响。用某些UNIX系统版本提供的系统调用跟踪特性，对ed操作进行跟踪分析发现，ed将新内容写到一个临时文件中，然后删除原文件，最后将临时文件名改为原文件名。强制性锁机制对unlink函数没有影响，于是这一切就发生了。

456

在Solaris中，用truss(1)命令可以得到一个进程的系统调用跟踪信息，在FreeBSD和Mac OS X中，则使用ktrace(1)和kdump(1)命令。Linux提供strace(1)命令跟踪进程的系统调用。

- 不能用vi编辑程序编辑该文件。vi可以读该文件，但是如果试图将新的数据写到该文件中，则出错返回（EAGAIN）。如果试图将新数据添加到该文件中，则write阻塞。vi的这种行为与所预料的一样。
- 使用Korn shell的>和>>运算符重写或添写到该文件中，产生出错信息“cannot create”。
- 在Bourne shell下使用>运算符出错，但是使用>>运算符则阻塞，在解除了强制性锁后再继续进行处理。（这两种shell在执行添加操作时会产生这样的区别，是因为Korn shell以O_CREAT和O_APPEND标志打开文件，上面已提及指定O_CREAT会产生出错返回，而Bourne shell在该文件已存在时并不指定O_CREAT，所以open成功，而下一个write则阻塞。）

产生的结果随所用操作系统版本的不同而变。从这样一个例子中可见，在使用强制性锁时还需有所警惕。从ed实例可以看到，强制性锁是可以设法避开的。

一个别有用心的用户可以对大家都可读的文件加一把读锁（强制性），这样就能阻止任何其他人写该文件（当然，该文件应当是强制性锁机制起作用的，这可能要求该用户能够更改该文件的权限位）。考虑一个数据库文件，它是大家都可读的，并且是强制性锁机制起作用的。如果一个别有用心的用户对该整个文件保有一把读锁，则其他进程不能再写该文件。

程序清单14-6用于确定一个系统是否支持强制性锁机制。

程序清单14-6 确定是否支持强制性锁

457

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int             fd;
    pid_t           pid;
    char            buf[5];
    struct stat     statbuf;
    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");

    /* turn on set-group-ID and turn off group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        /* write lock entire file */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("write_lock error");

        TELL_CHILD(pid);

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
    } else {                /* child */
        WAIT_PARENT();      /* wait for parent to set lock */

        set_fl(fd, O_NONBLOCK);

        /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1)    /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n",
          errno);

        /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buf, 2) < 0)
```

```
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buf = %2.2s\n",
              buf);
    }
    exit(0);
}
```

458

此程序首先创建一个文件，并使强制性锁机制对其起作用。然后程序分裂为父进程和子进程。父进程对整个文件设置一把写锁，子进程则将该文件的描述符设置为非阻塞的，然后企图对该文件设置一把读锁，我们期望这会出错返回，并希望看到系统返回是EACCES或EAGAIN。接着，子进程将文件读、写位置调整到文件起点，并试图读（read）该文件。如果系统提供强制性锁机制，则read应返回EACCES或EAGAIN（因为该描述符是非阻塞的），否则read返回所读的数据。在Solaris 9运行此程序（该系统支持强制性锁机制），得到：

```
$ ./a.out temp.lock
read_lock of already-locked region returns 11
read failed (mandatory locking works): Resource temporarily unavailable
```

查看系统头文件或intro(2)手册页，可以看到errno 11对应于EAGAIN。若在FreeBSD 5.2.1运行此程序，则得到：

```
$ ./a.out temp.lock
read_lock of already-locked region returns 35
read OK (no mandatory locking), buf = ab
```

其中，errno 35对应于EAGAIN。该系统不支持强制性锁。 □

实例

让我们回到本节的第一个问题：若两个人同时编辑同一个文件将会怎样呢？一般的UNIX系统文本编辑器并不使用记录锁，所以对此问题的回答仍然是：该文件的最后结果取决于写该文件的最后一个进程。

某些版本的vi编辑器使用建议性记录锁。即使我们正在使用这种版本的vi编辑器，但是它并不能阻止其他用户使用另一个没有使用建议性记录锁的编辑器。

若系统提供强制性记录锁，那么我们可以修改自己常用的编辑器（如果有该编辑器的源代码）。如没有该编辑器的源代码，那么可以试一试下述方法。编写一个vi的前端程序。该程序立即调用fork，然后父进程等待子进程终止，子进程打开在命令行中指定的文件，使强制性锁起作用，对整个文件设置一把写锁，然后执行vi。在vi运行时，该文件是加了写锁的，所以其他用户不能修改它。当vi结束时，父进程从wait返回，此时自编的前端程序也就结束了。

这种类型的前端程序是可以编写的，但却往往不能起作用。问题出在大多数编辑器通常在读完输入文件后关闭它。只要引用被编辑文件的描述符关闭了，那么加在该文件上的锁就被释放了。这意味着，编辑器读了该文件的内容后，随即关闭了该文件，那么锁也就不存在了。前端程序中没有任何方法可以阻止这一点。 □

第20章的数据库函数库使用了记录锁以提供多个进程的并发访问。该章也提供了定时测量数据，以观察记录锁对进程的影响。

459

14.4 STREAMS

STREAMS（流）是系统V提供的构造内核设备驱动程序和网络协议包的一种通用方法，对STREAMS进行讨论的目的是为了理解系统V的终端接口，I/O多路转接中poll（轮询）函数的使用（见14.5.2节），以及基于STREAMS的管道和命名管道的实现（见17.2节和17.2.1节）。

请注意不要将本章说明的STREAMS（流）与标准I/O库（见5.2节）中使用的流（stream）相混淆。流机制是由Dennis Ritchie开发的[Ritchie 1984]，其目的是用通用、灵活的方法改写传统的字符I/O系统（c-list）并与网络协议相适应，后来稍加增强，名称改用大写字母，成为STREAMS机制，被加入到SVR3。SVR4则提供了对STREAMS的全面支持（亦即，一个基于STREAMS的终端I/O系统）。[AT&T 1990d]对SVR4实现进行了说明。Rago[1993]讨论了用户层的STREAMS编程和内核层的STREAMS编程。

在Single UNIX Specification中，STREAMS是一种可选择的特征（XSI STREAMS Option Group）。在本书讨论的四种平台中，只有Solaris对STREAMS提供了很自然的支持。在Linux中，STREAMS子系统是可用的，但是用户必须自行将该子系统安装到系统中，通常它默认为不包括在系统中。

流在用户进程和设备驱动程序之间提供了一条全双工通路。流无需和实际硬件设备直接会话，流也可以用来构造伪设备驱动程序。图14-4示出了一个简单流（simple stream）的基本结构。

在流首（stream head）之下可以压入处理模块。这可以用ioctl命令实现。图14-5示出了包含一个处理模块的流。各方框之间用两根带箭头的线连接，以突出流的全双工特征，并强调两个方向的处理是相互独立进行的。

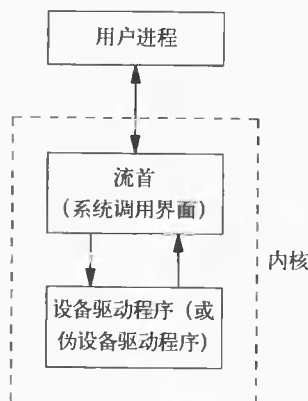


图14-4　一个简单流

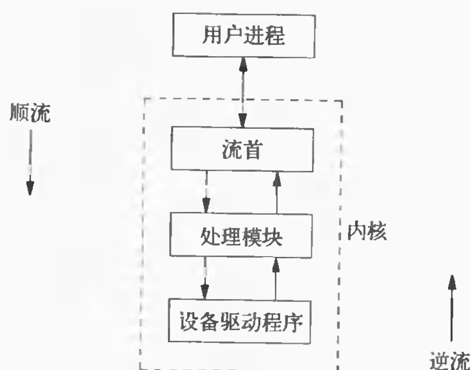


图14-5　具有处理模块的流

任意数量的处理模块可以压入流。我们使用术语压入，是因为每一新模块总是插到流首之下，而将以前的模块下压。（这类似于后进先出的栈。）图14-5标出了流的两侧，分别称为顺流（downstream）和逆流（upstream）。写到流首的数据将顺流而下传送，由设备驱动程序读到的数据则逆流向上传送。

460

STREAMS模块是作为内核的一部分执行的，这类似于设备驱动程序。当构造内核时，STREAMS模块联编进入内核。如果系统支持动态可装入的内核模块（Linux和Solaris是这样做的），则我们可以试图将没有联编进内核的STREAMS模块压入一个流；但不保证STREAMS模块和驱动程序的任意组合将能正常工作。

用第3章中说明的函数访问流，它们是：open、close、read、write和ioctl。另外，

在SVR3内核中增加了3个支持流的新函数（getmsg、putmsg和poll），在SVR4中又加了两个处理流内不同优先级波段消息的函数（getpmsg和putpmsg）。本节将说明这5个新函数。

打开（open）流时使用的路径名参数通常在/dev目录之下。仅仅用ls -l查看设备名，不能判断该设备是不是STREAMS设备。所有STREAMS设备都是字符特殊文件。

虽然某些有关STREAMS的文献暗示我们可以编写处理模块，并且不加细究地就可将它们压入流中，但是编写这些模块如同编写设备驱动程序一样，需要专门的技术。通常只有特殊的应用程序或函数才压入和弹出STREAMS模块。

461

在STREAMS之前，终端是用现存的c-list机制处理的。(Bach[1986]的10.3.1节和McKusick等[1996]的10.6节分别说明SVR2和4.4BSD中的c-list机制。)将基于字符的其他设备添加到内核中通常涉及编写设备驱动程序，并将所有有关部分都安排在驱动程序中。对新设备的访问典型地通过原始设备进行，这意味着每个用户的read，write最后都直通进入设备驱动程序。流机制使这种交互作用方式更加灵活且条理清晰，使得数据可以用STREAMS消息方式在流首和驱动程序之间传送，并使任意数的中间处理模块可对数据进行操作。

1. STREAMS消息

STREAMS的所有输入和输出都基于消息。流首和用户进程使用read、write、ioctl、getmsg、getpmsg、putmsg和putpmsg交换消息。在流首、各处理模块和设备驱动程序之间，消息可以顺流而下，也可以逆流而上。

在用户进程和流首之间，消息由下列几部分组成：消息类型、可选择的控制信息以及可选择的数据。表14-4列出了对应于write、putmsg和putpmsg的不同参数所产生的不同消息类型。控制信息和数据由strbuf结构指定：

```
struct strbuf
  int  maxlen;  /* size of buffer */
  int  len;     /* number of bytes currently in buffer */
  char *buf;    /* pointer to buffer */
};
```

当用putmsg或putpmsg发送消息时，len指定缓冲区中数据的字节数。当用getmsg或getpmsg接收消息时，maxlen指定缓冲区长度（使内核不会溢出缓冲区），而len则由内核设置为存放在缓冲区中的数据量。消息长度为0是允许的，len为−1说明没有控制信息或数据。

为什么需要传送控制信息和数据两者呢？提供这两者使我们可以实现用户进程和流之间的服务接口。Olander, McGrath和Israel[1986]说明了系统V服务接口的原先实现。AT&T[1990d]第5章详细说明了服务接口，还使用了一个简单的实例。可能最为人了解的服务接口是系统V的传输层接口（Transport Layer Interface，TLI），它提供了网络系统接口，Rago [1993]第4章对此进行了说明。

控制信息的另一个例子是发送一个无连接的网络消息（数据报）。为了发送该消息，需要说明消息的内容（数据）和该消息的目的地址（控制信息）。如果不能将数据和控制一起发送，那么就要某种专门设计的方案。例如，可以用ioctl说明地址，然后用write发送数据。另一种技术可能要求地址占用数据的前*N*个字节，而数据是用write写的。将控制信息与数据分开，并且提供处理两者的函数（putmsg和getmsg）是处理这种问题的较清晰的方法。

有约25种不同类型的消息，但是只有少数几种用于用户进程和流首之间，其余的只在内核中顺流、逆流传送。（对于编写流处理模块的人员而言，这些消息是非常有用的，但是对编写用户级代码的人员而言，它们可以忽略。）在我们所使用的函数（read、write、getmsg、getpmsg、putmsg和putpmsg）中，只涉及三种消息类型，它们是：

462

- M_DATA（I/O的用户数据）。
- M_PROTO（协议控制信息）。
- M_PCPROTO（高优先级协议控制信息）。

流中的消息都有一个排队优先级：

- 高优先级消息（最高优先级）。
- 优先级波段消息。
- 普通消息（最低优先级）。

普通消息是优先级波段为0的消息。优先级波段消息的波段可在1~255之间，波段愈高，优先级也愈高。高优先级消息的特殊性在于，在任何时刻流首只有一个高优先级消息排队。在流首读队列已有一个高优先级消息时，另外的高优先级消息会被丢弃。

表14-4　write、putmsg和putpmsg产生的STREAMS消息类型

函　数	控制？	数据？	波　段	标　志	产生的消息类型
write	N/A	是	N/A	N/A	M_DATA（普通）
putmsg	否	否	N/A	0	不发送消息，返回0
putmsg	否	是	N/A	0	M_DATA（普通）
putmsg	是	是或否	N/A	0	M_PROTO（普通）
putmsg	是	是或否	N/A	RS_HIPRI	M_PCPROTO（高优先级）
putmsg	否	是或否	N/A	RS_HIPRI	出错，EINVAL
putpmsg	是或否	是或否	0-255	0	出错，EINVAL
putpmsg	否	否	0-255	MSG_BAND	不发送消息，返回0
putpmsg	否	是	0	MSG_BAND	M_DATA（普通）
putpmsg	否	是	1-255	MSG_BAND	M_DATA（优先级波段）
putpmsg	是	是或否	0	MSG_BAND	M_PROTO（普通）
putpmsg	是	是或否	1-255	MSG_BAND	M_PROTO（优先级波段）
putpmsg	是	是或否	0	MSG_HIPRI	M_PCPROTO（高优先级）
putpmsg	否	是或否	0	MSG_HIPRI	出错，EINVAL
putpmsg	是或否	是或否	非零	MSG_HIPRI	出错，EINVAL

每个STREAMS模块有两个输入队列。一个接收来自它上面模块的消息，这种消息从流首向驱动程序顺流传送。另一个接收来自它下面模块的消息，这种消息从驱动程序向流首逆流传送。在输入队列中的消息按优先级从高到低排列。表14-4列出了针对write、putmsg和putpmsg的不同参数，产生不同优先级的消息。

有一些消息我们未加考虑。例如，若流首从它下面接收到M_SIG消息，则产生一信号。这种方法用于终端行规程模块向具有控制终端的前台进程组发送终端产生的信号。

2. putmsg和putpmsg函数

putmsg和putpmsg函数用于将STREAMS消息（控制信息或数据，或两者）写至流中。这两个函数的区别是后者允许对消息指定一个优先级波段。

```
#include <stropts.h>

int putmsg(int filedes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flag);

int putpmsg(int filedes, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flag);
```

两个函数返回值：若成功则返回0，若出错则返回-1

463

对流也可以使用write函数，它等效于不带任何控制信息、*flag*为0的putmsg。

这两个函数可以产生三种不同优先级的消息：普通、优先级波段和高优先级。表14-4详细列出了这两个函数中几个参数的各种可能组合，以及所产生的不同类型的消息。

在表14-4中，N/A表示不适用。消息控制列中的“否”对应于空*ctlptr*参数，或*ctlptr->len*为-1。该列中的“是”对应于*ctlptr*非空，以及*ctlptr->len*大于等于0。这些说明同样适用于消息的数据部分（用*dataptr*代替*ctlptr*）。

3. STREAMS ioctl操作

3.15节曾提到过ioctl函数，它能做其他I/O函数不处理的事情。STREAMS系统继承了这种传统。

在Linux和Solaris中，使用ioctl可对流执行将近40种不同的操作。其中大多数操作的说明请见streamio(7)手册页。头文件<stropts.h>应包括在使用这些操作的C代码中。ioctl的第二个参数*request*说明执行哪一个操作。所有*request*都以I_开始。第三个参数的作用与*request*有关，有时它是一个整型值，有时它是指向一个整型或一个数据结构的指针。

实例：isastream函数

有时需要判断一个描述符是否引用一个流。这与调用isatty函数来判断一个描述符是否引用一个终端设备相类似（见18.9节）。Linux和Solaris为此提供了isastream函数。

464

```
#include <stropts.h>

int isastream(int filedes);
```

返回值：若为STREAMS设备则返回1，否则返回0

与isatty类似，它通常是用一个只对STREAMS设备才有效的ioctl函数来进行测试的。程序清单14-7是该函数的一种可能的实现。它使用I_CANPUT ioctl来测试由第三个参数说明的优先级波段（本实例中为0）是否可写。如果该ioctl执行成功，则它对所涉及的流并未作任何改变。

程序清单14-7 检查描述符是否引用STREAMS设备

```
#include    <stropts.h>
#include    <unistd.h>

int
isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

程序清单14-8可用于测试此函数。

程序清单14-8 测试isastream函数

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
```

```
    int     i, fd;

    for (i = 1; i < argc; i++) {
        if ((fd = open(argv[i], O_RDONLY)) < 0) {
            err_ret("%s: can't open", argv[i]);
            continue;
        }

        if (isastream(fd) == 0)
            err_ret("%s: not a stream", argv[i]);
        else
            err_msg("%s: streams device", argv[i]);
    }

    exit(0);
}
```

465

在Solaris 9下运行此程序，得到很多由ioctl函数返回的出错信息：

```
$ ./a.out /dev/tty /dev/fb /dev/null /etc/motd
/dev/tty: streams device
/dev/fb: not a stream: Invalid argument
/dev/null: not a stream: No such device or address
/etc/motd: not a stream: Inappropriate ioctl for device
```

/dev/tty在Solaris之下是个STREAMS设备，这与我们所期望的一致。/dev/fb不是一个STREAMS设备，但它是支持其他ioctl请求的字符特殊文件。对于不知道这种ioctl请求的设备，它返回EINVAL。/dev/null是一种不支持任何ioctl操作的字符特殊文件，所以ioctl返回ENODEV。最后，/etc/motd是一个普通文件，而不是字符特殊文件，所以返回ENOTTY（这种情况下的经典返回值）。我们从未见到曾期望的出错信息ENOSTR（"Device is not a stream"）。

> ENOTTY的原意是"Not a typewriter"，它是个历史产物，当ioctl企图对并不引用字符特殊设备的描述符进行操作时，UNIX系统内核都返回ENOTTY。在Solaris中，该消息已被改为"Inappropriate ioctl for device."。

□

实　例

如果ioctl的参数*request*是I_LIST，则系统返回已压入该流所有模块的名字，包括最顶端的驱动程序。（指明最顶端的原因是，在多路转接驱动程序的情况下，有多个驱动程序。Rago [1993]第12章讨论了多路转接驱动程序的细节。）其第三个参数应当是指向str_list结构的指针。

```
struct str_list {
  int                sl_nmods;    /* number of entries in array */
  struct str_mlist  *sl_modlist;  /* ptr to first element of array */
};
```

应将sl_modlist设置为指向str_mlist结构数组的第一个元素，将sl_nmods设置为该数组中的项数：

```
struct str_mlist {
  char  l_name[FMNAMESZ+1];   /* null terminated module name */
};
```

常量FMNAMESZ在头文件<sys/conf.h>中定义，其值常常是8。l_name的实际长度是

FMNAMESZ+1，增加1个字节是为了存放null终止符。

如果ioctl的第三个参数是0，则该函数返回值是模块数，而不是模块名。我们将先用这种ioctl调用确定模块数，然后再分配所要求的str_mlist结构数。

程序清单14-9例示了I_LIST操作。由ioctl返回的名字列表并不对模块和驱动程序进行区分，但是考虑到该列表的最后一项是处于流底部的驱动程序，所以在打印时将其标明为驱动程序。 466

程序清单14-9 列表流中的模块名

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>
#include <sys/conf.h>

int
main(int argc, char *argv[])
{
    int             fd, i, nmods;
    struct str_list list;

    if (argc != 2)
        err_quit("usage: %s <pathname>", argv[0]);

    if ((fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if (isastream(fd) == 0)
        err_quit("%s is not a stream", argv[1]);

    /*
     * Fetch number of modules.
     */
    if ((nmods = ioctl(fd, I_LIST, (void *) 0)) < 0)
        err_sys("I_LIST error for nmods");
    printf("#modules = %d\n", nmods);

    /*
     * Allocate storage for all the module names.
     */
    list.sl_modlist = calloc(nmods, sizeof(struct str_mlist));
    if (list.sl_modlist == NULL)
        err_sys("calloc error");
    list.sl_nmods = nmods;

    /*
     * Fetch the module names.
     */
    if (ioctl(fd, I_LIST, &list) < 0)
        err_sys("I_LIST error for list");

    /*
     * Print the names.
     */
    for (i = 1; i <= nmods; i++)
        printf("  %s: %s\n", (i == nmods) ? "driver" : "module",
          list.sl_modlist++->l_name);

    exit(0);
}
```

467

为了弄清楚哪些STREAMS模块已压入控制终端，我们以网络登录和控制台登录两种方式

运行程序清单14-9所示程序，得到下列结果：

```
$ who
sar         console      May  1 18:27
sar         pts/7        Jul 12 06:53
$ ./a.out /dev/console
#modules = 5
  module: redirmod
  module: ttcompat
  module: ldterm
  module: ptem
  driver: pts
$ ./a.out /dav/pts/7
#modules = 4
  module: ttcompat
  module: ldterm
  module: ptem
  driver: pts
```

在这两种情形中，4个STREAMS模块都是一样的（ttcompat、ldterm、ptem和pts），唯一的区别是控制台在其流的顶部多了一个模块，它的作用是帮助虚控制台重定向。运行此程序的计算机在控制台上运行了一个窗口系统，所以/dev/console实际上引用的是伪终端而非硬连线设备。第19章将说明伪终端。 □

4. 写（write）至STREAMS设备

在表14-4中可以看到写至STREAMS设备产生一个M_DATA消息。一般情况确实如此，但是也还有一些细节需要考虑。首先，流中最顶部的一个处理模块规定了可顺流传送的最小、最大数据包长度（无法查询该模块中规定的这些值）。如果写的数据长度超过最大值，则流首将这一数据按最大长度分解成若干数据包。最后一个数据包的长度可能不到最大值。

接着要考虑的是：如果向流写0个字节，又将如何呢？除非流引用管道或FIFO，否则就顺流发送0长度消息。对于管道和FIFO，为与以前版本兼容，系统的默认处理方式是忽略0长度write。可以用ioctl设置管道和FIFO流的写模式，从而更改这种默认处理方式。

5. 写模式

可以用两个ioctl命令取得和设置一个流的写模式。如果将*request*设置为I_GWROPT，第三个参数设置为指向一个整型变量的指针，则该流的当前写模式在该整型量中返回。如果将*request*设置为I_SWROPT，第三个参数是一个整型值，则其值成为该流新的写模式。如同处理文件描述符标志和文件状态标志（见3.14节）一样,总是应当先取当前写模式值,然后修改它，而不只是将写模式设置为某个绝对值（很可能会关闭某些原来打开的位）。

468

目前，只定义了两个写模式值。

SNDZERO　对管道和FIFO的0长度write会造成顺流传送一个0长度消息。按系统默认，0长度写不发送消息。

SNDPIPE　在流上已出错后，若调用write或putmsg，则向调用进程发送SIGPIPE信号。

流也有读模式，我们先说明getmsg和getpmsg函数，然后再说明读模式。

6. getmsg和getpmsg函数

使用read、getmsg或getpmsg函数从流首读STREAMS消息。

```
#include <stropts.h>

int getmsg(int filedes, struct strbuf *restrict ctlptr,
           struct strbuf *restrict dataptr, int *restrict flagptr);

int getpmsg(int filedes, struct strbuf *restrict ctlptr,
            struct strbuf *restrict dataptr, int *restrict bandptr,
            int *restrict flagptr);
```

两个函数返回值：若成功则返回非负值，若出错则返回-1

注意，flagptr和bandptr是指向整型的指针。在调用之前，这两个指针所指向的整型单元中应设置成所希望的消息类型；在返回时，此整型量设置为所读到的消息的类型。

如果*flagptr*指向的整型单元的值是0，则getmsg返回流首读队列中的下一个消息。如果下一个消息是高优先级消息，则在返回时，*flagptr*所指向的整型单元设置为RS_HIPRI。如果希望只接收高优先级消息，则在调用getmsg之前必须将*flagptr*所指向的整型单元设置为RS_HIPRI。

getpmsg使用一个不同的常量集。为了只接收高优先级消息，我们可将*flagptr*指向的整型单元设置为MSG_HIPRI。为了只接收某个优先级波段或以上波段（包括高优先级消息）的消息，我们可将该整型单元设置为MSG_BAND，然后将*bandptr*指向的整型单元设置为该波段的非0优先级值。如果只希望接收第1个可用消息，则可将*flagptr*指向的整型单元设置为MSG_ANY；在返回时，该整型值将改写为MSG_HIPRI或MSG_BAND，这取决于接收到的消息的类型。如果取到的消息并非高优先级消息，那么*bandptr*指向的整型将包括消息的优先级波段值。

如果*clptr*是null，或*ctlptr->maxlen*是−1，那么消息的控制部分仍保留在流首读队列中，我们将不处理它。类似地，如果*dataptr*是null，或者*dataptr->maxlen*是-1，那么消息的数据部分仍保留在流首读队列中，我们也不处理它。否则，将按照缓冲区的容量取到消息中尽可能多的控制和数据部分，余下部分仍留在队首，等待下次取用。

469

如果*getmsg*和*getpmsg*调用取到一消息，那么返回值是0。如果消息控制部分中有一些余留在流首读队列中，那么返回常量MORECTL。类似地，如果消息数据中有一些余留在流首读队列中，那么返回常量MOREDATA。如果控制和数据都有一些余留在流首读队列中，那么返回常量值是（MORECTL|MOREDATA）。

7. 读模式

如果读（read）STREAMS设备会发生些什么呢？有两个潜在的问题：

(1) 如果读到流中消息的记录边界将会怎样？

(2) 如果调用read，而流中下一个消息有控制信息又将如何？

对第一种情况的默认处理模式称为字节流模式。read从流中取数据直至满足了所要求的字节数，或者已经不再有数据。在这种模式中，忽略流中消息的边界。对第二种情况的默认处理是，如果在队列的前端有控制消息，则read出错返回。可以改变这两种默认处理模式。

调用ioctl时，若将*request*设置为I_GRDOPT，第三个参数又是指向一个整型单元的指针，则对该流的当前读模式在该整型单元中返回。如果将*request*设置为I_SRDOPT，第三个参数是整型值，则将该流的读模式设置为该值。读模式值可由下列三个常量指定：

RNORM　　普通，字节流模式，如上所述这是默认模式。

RMSGN　　消息不丢弃模式。read从流中取数据直至读到所要求的字节数，或者到达消息边界。如果某次read只用了消息的一部分，则其余下部分仍留在流中，以

供下一次读。

RMSGD　　消息丢弃模式。这与不丢弃模式的区别是，如果某次读只用了消息的一部分，则余下部分就被丢弃，不再使用。

在读模式中还可指定另外三个常量，以便设置在读到流中包含协议控制信息的消息时read的处理方法：

RPROTNORM　协议–普通模式。read出错返回，errno设置为EBADMSG。这是默认模式。

RPROTDAT　协议–数据模式。read将控制部分作为数据返回给调用者。

470

RPROTDIS　协议–丢弃模式。read丢弃消息中的控制信息，但是返回消息中的数据。

任一时刻，只能设置一种消息读模式以及一种协议读模式。默认读模式是（RNORM | RPROTNORM)。

程序清单14-10是在程序清单3-3的基础上改写的，它用getmsg代替了read。

程序清单14-10　用getmsg将标准输入复制到标准输出

```
#include "apue.h"
#include <stropts.h>

#define BUFFSIZE    4096

int
main(void)
{
    int             n, flag;
    char            ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf   ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0;       /* return any message */
        if ((n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("getmsg error");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
          flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("write error");
    }
}
```

如果在Solaris（其管道和终端都是用STREAMS实现的）下运行此程序则得：

```
$ echo hello, world | ./a.out              要求基于STREAMS的管道
flag = 0, ctl.len = -1, dat.len = 13
hello, world
flag = 0, ctl.len = 0, dat.len = 0         表明STREAMS挂断
$ ./a.out                                  要求基于STREAMS的终端
this is line 1
flag = 0, ctl.len = -1, dat.len = 15
```

471

```
this is line 1
and line 2
flag = 0, ctl.len = -1, dat.len = 11
and line 2
^D                                            键入终端EOF字符
flag = 0, ctl.len = -1, dat.len = 0           tty文件结尾与挂断不相同
$ ./a.out < /etc/motd
getmsg error: Not a stream device
```

当管道被关闭时（当echo终止时），它对程序清单14-10表现为一个STREAMS挂断，控制长度和数据长度都设置为0。（15.2节将讨论管道。）但是对于终端，键入文件结束字符只使返回的数据长度为0。这与STREAMS挂断并不相同。如所预料的一样，将标准输入重新定向到一个非STREAMS设备，getmsg出错返回。 □

14.5 I/O多路转接

当从一个描述符读，然后又写到另一个描述符时，可以在下列形式的循环中使用阻塞I/O：

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");
```

这种形式的阻塞I/O到处可见。但是如果必须从两个描述符读，又将如何呢？如果仍旧使用阻塞I/O，那么就可能长时间阻塞在一个描述符上，而另一个描述符虽有很多数据却不能得到及时处理。所以为了处理这种情况显然需要另一种不同的技术。

让我们观察telnet(1)命令的结构。该程序读终端（标准输入），将所得数据写到网络连接上；同时读网络连接，将所得数据写到终端上（标准输出）。在网络连接的另一端，telnetd守护进程读用户在终端上所键入的内容，并将其送给shell，这如同用户登录在远程机器上一样。telnetd守护进程将执行用户键入命令，而产生的输出通过telnet命令送回给用户，并显示在用户终端上。图14-6显示这种工作情景。



图14-6 telnet程序概观

telnet进程有两个输入、两个输出。对这两个输入中的任一个都不能使用阻塞read，因为我们永远不知道哪一个输入有我们需要的数据。 472

处理这种特殊问题的一种方法是，用fork将一个进程变成两个进程，每个进程处理一条数据通路。图14-7中显示了这种安排。（系统V uucp通信包提供了cu(1)命令，其结构与此相似。）

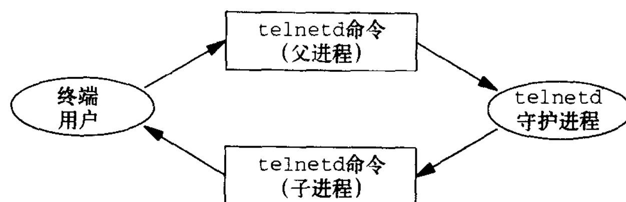


图14-7 使用两个进程实现telnet程序

如果使用两个进程，则可使每个进程都执行阻塞read。但是这也产生了问题：操作什么时候终止？如果子进程接收到文件结束符telnetd守护进程使网络连接断开，那么该子进程终止，然后父进程接收到SIGCHLD信号。但是，如若父进程终止（用户在终端上键入了文件结束符），那么父进程应通知子进程停止。为此可以使用一个信号（例如SIGUSR1），但这使程序变得更加复杂。

我们可以不使用两个进程，而是用一个进程中的两个线程。这避免了终止的复杂性，但却要求处理线程之间的同步，在减少复杂性方面这可能会是得不偿失。

另一个方法是仍旧使用一个进程执行该程序，但使用非阻塞I/O读取数据。基本方法是将两个输入描述符都设置为非阻塞的，对第一个描述符发一个read。如果该输入上有数据，则读数据并处理它；如果无数据可读，则read立即返回。然后对第二个描述符作同样的处理。在此之后，等待若干秒，然后再读第一个描述符。这种形式的循环称为*轮询*（polling）。这种方法的不足之处是浪费CPU时间。因为大多数时间实际上是无数据可读的，但是仍花费时间不断反复执行read系统调用。在每次循环后要等多长时间再执行下一轮循环也很难确定。虽然轮询技术在支持非阻塞I/O的系统上都可使用，但是在多任务系统中应当避免使用这种方法。

还有一种技术称之为*异步I/O*（asynchronous I/O）。其基本思想是进程告诉内核，当一个描述符已准备好可以进行I/O时，用一个信号通知它。这种技术有两个问题。第一，并非所有系统都支持这种机制（在Single UNIX Specification中这是一个可选择的设施）。系统V为此技术提供了SIGPOLL信号，但是仅当描述符引用STREAMS设备时，此信号才能工作。BSD有一个类似的信号SIGIO，但也有类似的限制，仅当描述符引用终端设备或网络时才能工作。其次，这种信号对每个进程而言只有1个（SIGPOLL或SIGIO）。如果使该信号对两个描述符都起作用（在我们正在讨论的实例中，从两个描述符读），那么在接到此信号时进程无法判别是哪一个描述符已准备好可以进行I/O。为了确定是哪一个，仍需将这两个描述符都设置为非阻塞的，并顺序试执行I/O。14.6节将简要说明异步I/O。

473

一种比较好的技术是使用*I/O多路转接*（I/O multiplexing）。先构造一张有关描述符的列表，然后调用一个函数，直到这些描述符中的一个已准备好进行I/O时，该函数才返回。在返回时，它告诉进程哪些描述符已准备好可以进行I/O。

poll、pselect和select这三个函数使我们能够执行I/O多路转接。表14-5摘要列出了哪些平台支持这些函数。注意基本POSIX.1标准定义了select函数，而poll则是对该基本部分的XSI扩展。

表14-5　多种UNIX系统支持的I/O多路转接

系　统	poll	pselect	select	<sys/select.h>
SUS	XSI	•	•	•
FreeBSD 5.2.1	•	•	•	
Linux 2.4.22	•	•	•	•
Mac OS X 10.3	•	•	•	
Solaris 9	•		•	•

POSIX指定，为了在程序中使用select，必须包括<sys/select.h>。但是历史上，为了在程序中使用select，还要包括另外三个头文件，而且某些实现至今还落在标准之后。为此，要查看select手册页，弄清楚你所用的系统对它支持到何种程度。较老的系统要求在程序中包括<sys/types.h>、<sys/time.h>和<unistd.h>。

I/O多路转接在4.2 BSD中是用select函数提供的。虽然该函数主要用于终端I/O和网络I/O，但它对其他描述符同样是起作用的。SVR3在增加STREAMS机制时增加了poll函数，但一开始poll只对STREAMS设备起作用。SVR4支持对任一描述符起作用的poll。

14.5.1 select和pselect函数

在所有依从POSIX的平台上，select函数使我们可以执行I/O多路转接。传向select的参数告诉内核：

- 我们所关心的描述符。
- 对于每个描述符我们所关心的状态。（是否读一个给定的描述符？是否想写一个给定的描述符？是否关心一个描述符的异常状态？）
- 愿意等待多长时间（可以永远等待，等待一个固定量时间，或完全不等待）。

从select返回时，内核告诉我们：

- 已准备好的描述符的数量。
- 对于读、写或异常这三个状态中的每一个，哪些描述符已准备好。

474

使用这些返回信息，就可调用相应的I/O函数（一般是read或write），并且确知该函数不会阻塞。

```
#include <sys/select.h>

int select(int maxfdp1, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict exceptfds,
           struct timeval *restrict tvptr);
```

返回值：准备就绪的描述符数，若超时则返回0，若出错则返回-1

先说明最后一个参数，它指定愿意等待的时间：

```
struct timeval {
  long  tv_sec;    /* seconds */
  long  tv_usec;   /* and microseconds */
};
```

有三种情况：

tvptr==NULL

永远等待。如果捕捉到一个信号则中断此无限期等待。当所指定的描述符中的一个已准备好或捕捉到一个信号则返回。如果捕捉到一个信号，则select返回-1，errno设置为EINTR。

tvptr->tv_sec==0 && *tvptr->tv_usec*==0

完全不等待。测试所有指定的描述符并立即返回。这是得到多个描述符的状态而不阻塞select函数的轮询方法。

tvptr->tv_sec !=0 || *tvptr->tv_usec*!=0

等待指定的秒数和微秒数。当指定的描述符之一已准备好，或当指定的时间值已经超过时立即返回。如果在超时时还没有一个描述符准备好，则返回值是0（如果系统不提供微秒分辨率，则*tvptr->tv_usec*值取整到最近的支持值）。与第一种情况一样，这种等待可被捕捉到的信号中断。

> POSIX.1允许在实现中修改timeval结构中的值，所以在select返回后，你不能指望该结构仍旧保持调用select之前它所包含的值。FreeBSD 5.2.1、Mac OS X 10.3和Solaris 9都保持该结构中的值不变。但是Linux 2.4.22中，若在该时间值尚未超过时select就返回，那么将用余留时间值更新该结构。

中间三个参数*readfds*、*writefds*和*exceptfds*是指向描述符集的指针。这三个描述符集说明了我们关心的可读、可写或处于异常条件的各个描述符。每个描述符集存放在一个fd_set数据类型中。这种数据类型为每一可能的描述符保持了一位，其实现可如图14-8中所示。

475

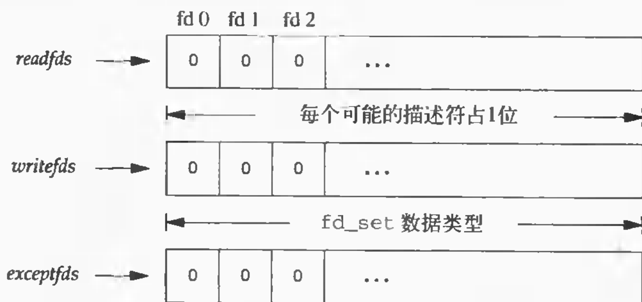


图14-8　对select指定读、写和异常条件描述符

对fd_set数据类型可以进行的处理是：分配一个这种类型的变量；将这种类型的一个变量值赋予同类型的另一个变量；或对于这种类型的变量使用下列四个函数中的一个。

```
#include <sys/select.h>

int FD_ISSET(int fd, fd_set *fdset);

                    返回值：若fd在描述符集中则返回非0值，否则返回0

void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

这些接口可实现为宏或函数。调用FD_ZERO将一个指定的fd_set变量的所有位设置为0。调用FD_SET设置一个fd_set变量的指定位。调用FD_CLR则将一指定位清除。最后，调用FD_ISSET测试一指定位是否设置。

声明了一个描述符集后，必须用FD_ZERO清除其所有位，然后在其中设置我们关心的各个位。这种操作序列如下所示：

```
fd_set   rset;
int      fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

从select返回时，用FD_ISSET测试该集中的一个给定位是否仍旧设置：

```
if (FD_ISSET(fd, &rset)) {
    ...
}
```

476

select的中间三个参数（指向描述符集的指针）中的任意一个或全部都可以是空指针，

这表示对相应状态并不关心。如果所有三个指针都是空指针，则select提供了较sleep更精确的计时器。（回忆10.19节，sleep等待整数秒，而对于select，其等待的时间可以小于1s；其实际分辨率取决于系统时钟。）习题14.6给出了这样一个函数。

select的第一个参数*maxfdp1*的意思是“最大描述符加1”。在三个描述符集中找出最大描述符编号值，然后加1，这就是第一个参数值。也可将第一个参数设置为FD_SETSIZE，这是<sys/select.h>中的一个常量，它说明了最大的描述符数（经常是1 024）。但是对大多数应用程序而言，此值太大了，多数应用程序只使用3~10个描述符。（某些应用程序使用更多的描述符，但这种UNIX程序并不具代表性。）如果将第三个参数设置为我们所关注的最大描述符编号值加1，内核就只需在此范围内寻找打开的位，而不必在三个描述符集中的数百位内搜索。

例如，若编写下列代码：

```
fd_set  readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

那么，图14-9显示了这两个描述符集的情况。

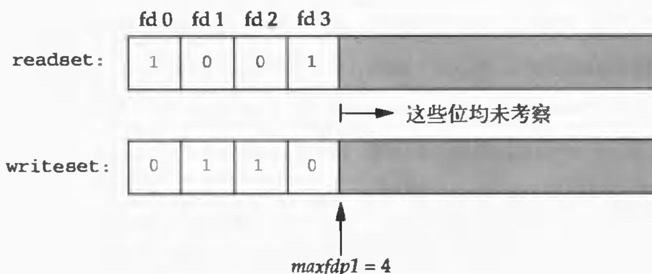


图14-9 select的示例描述符集

因为描述符编号从0开始，所以要在最大描述符编号值上加1。第一个参数实际上是要检查的描述符数（从描述符0开始）。

select有三个可能的返回值。

(1) 返回值−1表示出错。出错是有可能的，例如在所指定的描述符都没有准备好时捕捉到一个信号。在此种情况下，将不修改其中任何描述符集。

477

(2) 返回值0表示没有描述符准备好。若指定的描述符都没有准备好，而且指定的时间已经超过，则发生这种情况。此时，所有描述符集皆被清0。

(3) 正返回值表示已经准备好的描述符数，该值是三个描述符集中已准备好的描述符数之和，所以如果同一描述符已准备好读和写，那么在返回值中将其计为2。在这种情况下，三个描述符集中仍旧打开的位对应于已准备好的描述符。

对于“准备好”的意思要作一些更具体的说明：

- 若对读集（*readfds*）中的一个描述符的read操作将不会阻塞，则此描述符是准备好的。
- 若对写集（*writefds*）中的一个描述符的write操作将不会阻塞，则此描述符是准备好的。
- 若异常状态集（*exceptfds*）中的一个描述符有一个未决异常状态，则此描述符是准备好的。

现在，异常状态包括(a) 在网络连接上到达的带外数据，或者(b)在处于数据包模式的伪终端上发生了某些状态。(Stevens[1990]的15.10节中说明了后一种状态。)

- 对于读、写和异常状态，普通文件描述符总是返回准备好。

应当理解，一个描述符阻塞与否并不影响select是否阻塞。也就是说，如果希望读一个非阻塞描述符，并且以超时值为5s调用select，则select最多阻塞5s。相类似地，如果指定一个无限的超时值，则在该描述符数据准备好或捕捉到一个信号之前，select一直阻塞。

如果在一个描述符上碰到了文件结尾处，则select认为该描述符是可读的。然后调用read，它返回0，这是UNIX系统指示到达文件结尾处的方法。(很多人错误地认为，当到达文件结尾处时，select会指示一个异常状态。)

POSIX.1也定义了一个select的变体，它被称为pselect。

```
#include <sys/select.h>

int pselect(int maxfdp1, fd_set *restrict readfds,
            fd_set *restrict writefds, fd_set *restrict exceptfds,
            const struct timespec *restrict tsptr,
            const sigset_t *restrict sigmask);
```

返回值：准备就绪的描述符数，若超时则返回0，若出错则返回-1

除下列几点外，pselect与select相同：

- select的超时值用timeval结构指定，但pselect使用timespec结构。(回忆11.6节中timespec结构的定义。) timespec结构以秒和纳秒表示超时值，而非秒和微秒。如果平台支持这样精细的粒度，那么timespec就提供了更精准的超时时间。
- pselect的超时值被声明为const，这保证了调用pselect不会改变此值。
- 对于pselect可使用一可选择的信号屏蔽字。若*sigmask*为空，那么在与信号有关的方面，pselect的运行状况和select相同。否则，*sigmask*指向一信号屏蔽字，在调用pselect时，以原子操作的方式安装该信号屏蔽字。在返回时恢复以前的信号屏蔽字。

478

14.5.2 poll函数

poll函数类似于select，但是其程序员接口则有所不同。我们将会看到，虽然poll函数可用于任何类型的文件描述符，但它起源于系统V，所以poll与STREAMS系统紧紧相关。

```
#include <poll.h>

int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

返回值：准备就绪的描述符数，若超时则返回0，若出错则返回-1

与select不同，poll不是为每个状态（可读性、可写性和异常状态）构造一个描述符集，而是构造一个pollfd结构数组，每个数组元素指定一个描述符编号以及对其所关心的状态。

```
struct pollfd {
  int    fd;       /* file descriptor to check, or <0 to ignore */
  short  events;   /* events of interest on fd */
  short  revents;  /* events that occurred on fd */
};
```

*fdarray*数组中的元素数由*nfds*说明。

由于历史原因，声明*nfds*参数有几种不同的方式。SVR3说明*nfds*的类型为unsigned long，这似乎是太大了。在SVR4手册[AT&T 1990d]中，poll原型的第二个参数的数据类型为size_t（见表2-16中的基本系统数据类型）。但在<poll.h>包含的实际原型中，第二个参数的数据类型仍说明为unsigned long。Single UNIX Specification定义了新类型nfds_t，该类型允许实现选择对其合适的类型并且隐藏了应用细节。注意，因为返回值表示数组中满足事件（events）的项数，所以这种类型必须大得足以保持一个整型。

SVR4的SVID[AT&T1989]说明poll的第一个参数是struct pollfd *fdarray*[ ]，而SVR4手册页[AT&T 1990 d]则说明该参数为struct pollfd **fdarray*。在C语言中，这两种说明是等价的。我们使用第一种说明以重申*fdarray*指向一个结构数组，而不是指向单个结构的指针。

479

应将每个数组元素的events成员设置为表14-6中所示的值。通过这些值告诉内核我们对该描述符关心的是什么。返回时，内核设置revents成员，以说明对于该描述符已经发生了什么事件。（注意，poll没有更改events成员，这与select不同，select修改其参数以指示哪一个描述符已准备好了。）

表14-6 poll的events和revents标志

标志名	输入至events?	从revents得到结果?	说明
POLLIN	•	•	不阻塞地可读除高优先级外的数据（等效于POLLRDNORM|POLLRDBAND）
POLLRDNORM	•	•	不阻塞地可读普通数据（优先级波段为0）
POLLRDBAND	•	•	不阻塞地可读非0优先级波段数据
POLLPRI	•	•	不阻塞地可读高优先级数据
POLLOUT	•	•	不阻塞地可写普通数据
POLLWRNORM	•	•	与POLLOUT相同
POLLWRBAND	•	•	不阻塞地可写非0优先级波段数据
POLLERR		•	已出错
POLLHUP		•	已挂断
POLLNVAL		•	描述符不引用一打开文件

表14-6中头四行测试可读性，接着三行测试可写性，最后三行则是测试异常状态。最后三行是由内核在返回时设置的。即使在events字段中没有指定这三个值，如果相应条件发生，则在revents中也返回它们。

当一个描述符被挂断（POLLHUP）后，就不能再写向该描述符。但是仍可能从该描述符读取到数据。

poll的最后一个参数说明我们愿意等待多少时间。如同select一样，有三种不同的情形：

timeout == -1　永远等待。（某些系统在<stropts.h>中定义了常量INFTIM，其值通常是-1。）当所指定的描述符中的一个已准备好，或捕捉到一个信号时则返回。如果捕捉到一个信号，则poll返回-1，errno设置为EINTR。

timeout == 0　不等待。测试所有描述符并立即返回。这是得到很多个描述符的状态而不阻塞poll函数的轮询方法。

timeout > 0　等待*timeout*毫秒。当指定的描述符之一已准备好，或指定的时间值已超过时立即返回。如果已超时但是还没有一个描述符准备好，则返回值是0。（如果系统不提供毫秒分辨率，则*timeout*值取整到最近的支持值。）

480

应当理解文件结束与挂断之间的区别。如果正从终端输入数据，并键入文件结束字符，POLLIN被打开，于是就可读文件结束指示（read返回0）。POLLHUP在revents中没有打开。如果正在读调制解调器，并且电话线已挂断，则在revents中将接到POLLHUP通知。

与select一样，不论一个描述符是否阻塞，都不影响poll是否阻塞。

select和poll的可中断性

中断的系统调用的自动再启动是由4.2BSD引进的（见10.5节），但当时select函数是不再启动的。这种特性在大多数系统中一直延续了下来，即使指定了SA_RESTART也是如此。但是，在SVR4之下，如果指定了SA_RESTART，那么select和poll也是自动再启动的。为了在将软件移植到SVR4派生的系统上时防止这一点，如果信号可能中断对select或poll的调用，则总是使用signal_intr函数（见程序清单10-13）。

> 本书说明的各种实现在接到一信号时都不重启动poll和select，即使使用了SA_RESTART标志也是如此。

14.6 异步I/O

使用上一节说明的select和poll可以实现异步形式的通知。关于描述符的状态,系统并不主动告诉我们任何信息,我们需要进行查询（调用select或poll）。如在第10章中所述，信号机构提供一种以异步形式通知某种事件已发生的方法。由BSD和系统V派生的所有系统提供了使用一个信号（在系统V中是SIGPOLL，在BSD中是SIGIO）的异步I/O方法，该信号通知进程某个描述符已经发生了所关心的某个事件。

> 我们已了解到select和poll对任意描述符都能工作。但是关于异步I/O却有限制。在系统V派生的系统中，异步I/O只对STREAMS设备和STREAMS管道起作用。在BSD派生的系统中，异步I/O只对终端和网络起作用。

异步I/O的一个限制是每个进程只有一个信号。如果要对几个描述符进行异步I/O，那么在进程接收到该信号时并不知道这一信号对应于哪一个描述符。

> Single UNIX Specification包括一个可选择的通用异步I/O机制，这取自实时草案标准。它与本节所说明的机制无关。该机制解决了老的异步I/O机制存在的很多限制问题，但在此处不作进一步讨论。

14.6.1 系统V异步I/O

在系统V中，异步I/O是STREAMS系统的一部分。它只对STREAMS设备和STREAMS管道起作用。系统V的异步I/O信号是SIGPOLL。

481

为了对一个STREAMS设备启动异步I/O，需要调用ioctl，它的第二个参数（*request*）是I_SETSIG。第三个参数是由表14-7中的常量构成的整型值。这些常量在<stropts.h>中定义。

表14-7中“已到达”的意思是“已到达流首的读队列”。

除了调用ioctl说明产生SIGPOLL信号的条件以外，还应为该信号建立信号处理程序。回忆表10-1，对于SIGPOLL的默认动作是终止该进程，所以应当在调用ioctl之前建立信号处理程序。

表14-7 产生SIGPOLL信号的条件

常 量	说 明
S_INPUT	非高优先级消息已到达
S_RDNORM	普通消息已到达
S_RDBAND	非0优先级波段消息已到达
S_BANDURG	若此常量和S_RDBAND一起指定，则当一非0优先级波段消息已到达时，产生SIGURG信号而非SIGPOLL
S_HIPRI	高优先级消息已到达
S_OUTPUT	写队列不再满
S_WRNORM	与S_OUTPUT相同
S_WRBAND	可发送非0优先级波段消息
S_MSG	包含SIGPOLL信号的STREAMS信号消息已到达
S_ERROR	M_ERROR 消息已到达
S_HANGUP	M_HANGUP消息已到达

14.6.2 BSD异步 I/O

在BSD派生的系统中，异步I/O是SIGIO和SIGURG两个信号的组合。前者是通用异步I/O信号，后者则只用来通知进程在网络连接上到达了带外的数据。

为了接收SIGIO信号，需执行下列三步：

(1) 调用signal或sigaction为SIGIO信号建立信号处理程序。

(2) 以命令F_SETOWN（见3.14节）调用fcntl来设置进程ID和进程组ID，它们将接收对于该描述符的信号。

(3) 以命令F_SETFL调用fcntl设置O_ASYNC文件状态标志，使在该描述符上可以进行异步I/O（见表3-3）。

第(3)步仅能对指向终端或网络的描述符执行，这是BSD异步I/O设施的一个基本限制。

对于SIGURG信号，只需执行第(1)步和第(2)步。该信号仅对引用支持带外数据的网络连接描述符而产生。

482

14.7 readv和writev函数

readv和writev函数用于在一次函数调用中读、写多个非连续缓冲区。有时也将这两个函数称为**散布读**（scatter read）和**聚集写**（gather write）。

```
#include <sys/uio.h>

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);

ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);

                两个函数返回值：若成功则返回已读、写的字节数，若出错则返回-1
```

这两个函数的第二个参数是指向iovec结构数组的一个指针：

```
struct iovec {
  void   *iov_base;  /* starting address of buffer */
  size_t  iov_len;   /* size of buffer */
};
```

*iov*数组中的元素数由*iovcnt*说明。其最大值受限于IOV_MAX（参见表2-10）。图14-10显示了readv和writev的参数和iovec结构。

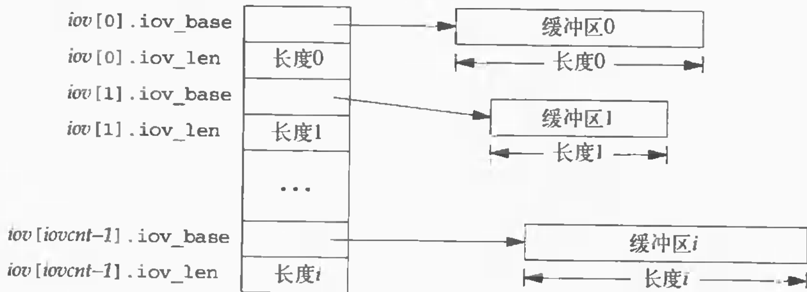


图14-10 readv和writev的iovec结构

writev以顺序*iov[0]*, *iov[1]*至*iov[iovcnt-1]*从缓冲区中聚集输出数据。writev返回输出的字节总数，通常，它应等于所有缓冲区长度之和。

readv则将读入的数据按上述同样顺序散布到缓冲区中。readv总是先填满一个缓冲区，然后再填写下一个。readv返回读到的总字节数。如果遇到文件结尾，已无数据可读，则返回0。

> 这两个函数始于4.2BSD，后来SVR4也提供它们。在Single UNIX Specification的XSI扩展中包括了这两个函数。
>
> 虽然Single UNIX Specification将缓冲区地址定义为void *类型，但在该标准前就已存在的很多实现仍使用char *。

483

实 例

在20.8节的_db_writeidx函数中，需将两个缓冲区内容连续地写到一个文件中。第二个缓冲区是调用者传递过来的一个参数，第一个缓冲区是我们创建的，它包含了第二个缓冲区的长度以及在文件中其他信息的偏移量。有三种方法可以实现这一要求：

(1) 调用write两次，一次一个缓冲区。

(2) 分配一个大到足以包含两个缓冲区的新缓冲区。将两个缓冲区的内容复制到新缓冲区中。然后对该缓冲区调用write一次。

(3) 调用writev输出两个缓冲区。

20.8节中使用了writev，但是将它与另外两种方法进行比较，对我们是很有启发的。表14-8显示了上面所述三种方法的结果。

表14-8 比较writev和其他技术所得的时间结果

操 作	Linux (Intel x86)			Mac OS X (PowerPC)		
	用户	系统	时钟	用户	系统	时钟
二次write	1.29	3.15	7.39	1.60	17.40	19.84
复制缓冲区，然后一次write	1.03	1.98	6.47	1.10	11.09	12.54
一次writev	0.70	2.72	6.41	0.86	13.58	14.72

所用的测试程序输出100字节的头文件，接着又输出200字节的数据。这样做1 048 576次，产生了一个300MB的文件。该程序按上面描述的3种方法分别编写了3个版本。使用times

（8.16节）测得它们在写操作前、后各使用的用户CPU时间、系统CPU时间和时钟时间。它们的单位都是秒。

正如我们所预料的，调用write两次的系统时间较调用write或writev一次要长，这与表3-2的结果类似。

接着要注意的是，在缓冲区复制后跟随一个write所用的CPU时间（用户加系统）要少于调用writev一次所耗费的CPU时间。对于单一write情况，我们先将用户层次的两个缓冲区复制至一个中间缓冲区，然后当调用write时内核将该中间缓冲区中的数据复制至其内部缓冲区。对于writev的情况，因为内核只需将数据直接复制进其内部缓冲区，所以复制工作应当少一些。但是，对于这种少量数据，使用writev的固定开销大于得益。随着需复制数据的增加，程序中复制缓冲区的开销也会增多。此时，writev这种替代方法就会有更大的吸引力。

注意不要依据表14-8中的数字对Linux和MacOS X之间的相对性能作过多的推断。这两种计算机有很大差别。它们有不同的处理器结构、不同量的RAM以及不同速度的磁盘。为了进行操作系统之间的比较，需要对每一种操作系统都使用相同的硬件。

484

总之，应当用尽量少的系统调用次数来完成任务。如果只写少量的数据，会发现自己复制数据然后使用一次write会比用writev更合算。但也可能发现，这样获得的性能提升并不值得，因为管理中间缓冲区会增加程序的复杂度。

14.8 readn和writen函数

管道、FIFO以及某些设备,特别是终端、网络和STREAMS设备有下列两种性质：

(1) 一次read操作所返回的数据可能少于所要求的数据，即使还没达到文件尾端也可能是这样。这不是一个错误，应当继续读该设备。

(2) 一次write操作的返回值也可能少于指定输出的字节数。这可能是由若干因素造成的，例如，下游模块的流量控制限制。这也不是错误，应当继续写余下的数据至该设备。（通常，只有对非阻塞描述符，或捕捉到一个信号时，才发生这种write的中途返回。）

在读、写磁盘文件时从未见到过这种情况，除非文件系统用完了空间，或者我们接近了配额限制，而不能将要求写的数据全部写出。

通常当读、写一个管道、网络设备或终端时，我们需要考虑这些特性。下面两个函数readn和writen的功能是读、写指定的N字节数据，并处理返回值小于要求值的情况。这两个函数只是按需多次调用read和write直至读、写了N字节数据。

```
#include "apue.h"

ssize_t readn(int filedes, void *buf, size_t nbytes);

ssize_t writen(int filedes, void *buf, size_t nbytes);
```

两个函数返回值：已读、写字节数，若出错则返回−1

类似于本书很多实例所使用的出错处理例程，我们定义这两个函数的目的是便于在后面实例中使用。readn和writen函数并非任何标准的组成部分。

在要将数据写到上面提到的文件类型上时，就可调用writen，但是只有当事先就知道要接收数据的数量时，才调用readn（通常只调用read接收来自这些设备的数据）。程序清单14-11

包含了writen和readn的一种实现，在后面的实例中，我们将使用它们。

程序清单14-11　readn和writen函数

```
#include "apue.h"

ssize_t             /* Read "n" bytes from a descriptor  */
readn(int fd, void *ptr, size_t n)
{
    size_t      nleft;
    ssize_t     nread;

    nleft = n;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break;      /* error, return amount read so far */
        } else if (nread == 0) {
            break;          /* EOF */
        }
        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}

ssize_t             /* Write "n" bytes to a descriptor  */
writen(int fd, const void *ptr, size_t n)
{
    size_t      nleft;
    ssize_t     nwritten;

    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break;      /* error, return amount written so far */
        } else if (nwritten == 0) {
            break;
        }
        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n - nleft);      /* return >= 0 */
}
```

注意，若在已经读、写了一些数据后出错，则这两个函数返回已传输的数据量，而非出错返回。与此类似，在读时如达到文件尾，而且在此之前已成功地读了一些数据，但尚未满足所要求的量，则readn返回已复制到调用者缓冲区中的字节数。

485～486

14.9　存储映射I/O

存储映射I/O（Memory-mapped I/O）使一个磁盘文件与存储空间中的一个缓冲区相映射。于是当从缓冲区中取数据，就相当于读文件中的相应字节。与此类似，将数据存入缓冲区，则

相应字节就自动地写入文件。这样就可以在不使用read和write的情况下执行I/O。

存储映射I/O伴随虚拟存储系统已经用了很多年。4.1BSD（1981）以其vread和vwrite函数提供了一种不同形式的存储映射I/O。4.2BSD没有使用这两个函数，而是企图换成mmap函数。但是由于McKusick et al. [1996] 2.5节中说明的理由，4.2BSD实际上并没有包含mmap函数。Gingell,Moran和Shannon[1987]说明了mmap的一种实现。现在，Single UNIX Specification存储映射文件选项中包括了mmap函数，在遵循XSI的系统中则应包含此函数。大多数UNIX系统都支持mmap函数。

为了使用这种功能，应首先告诉内核将一个给定的文件映射到一个存储区域中。这是由mmap函数实现的。

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flag, int filedes,
           off_t off);
```

返回值：若成功则返回映射区的起始地址，若出错则返回MAP_FAILED

*addr*参数用于指定映射存储区的起始地址。通常将其设置为0，这表示由系统选择该映射区的起始地址。此函数的返回地址是该映射区的起始地址。

*filedes*指定要被映射文件的描述符。在映射该文件到一个地址空间之前，先要打开该文件。*len*是映射的字节数。*off*是要映射字节在文件中的起始偏移量（下面将说明对*off*值有某些限制）。

*prot*参数说明对映射存储区的保护要求，见表14-9。

表14-9　映射存储区的保护要求

prot	说　明
PROT_READ	映射区可读
PROT_WRITE	映射区可写
PROT_EXEC	映射区可执行
PROT_NONE	映射区不可访问

可将*prot*参数指定为PROT_NONE，或者是PROT_READ、PROT_WRITE、PROT_EXEC任意组合的按位或。对指定映射存储区的保护要求不能超过文件open模式访问权限。例如，若该文件是只读打开的，那么对映射存储区就不能指定PROT_WRITE。

在说明*flag*参数之前，先看一下存储映射文件的基本情况。图14-11显示了一个存储映射文件。（见图7-3中进程存储空间的典型安排情况。）在此图中，“起始地址”是mmap的返回值。映射存储区位于堆和栈之间，这属于实现细节，各种实现之间可能不尽相同。

487

*flag*参数影响映射存储区的多种属性：

MAP_FIXED　返回值必须等于addr。因为这不利于可移植性，所以不鼓励使用此标志。如果未指定此标志，而且addr非0，则内核只把addr视为在何处设置映射区的一种建议，但是不保证会使用所要求的地址。将addr指定为0可获得最大可移植性。

在遵循POSIX的系统中，对MAP_FIXED的支持是可选择的，但遵循XSI的系统则要求支持MAP_FIXED。

MAP_SHARED　这一标志说明了本进程对映射区所进行的存储操作的配置。此标志指定

存储操作修改映射文件，也就是说，存储操作相当于对该文件的write。必须指定本标志或下一个标志（MAP_PRIVATE），但不能同时指定两者。

488

MAP_PRIVATE　本标志说明，对映射区的存储操作导致创建该映射文件的一个私有副本。所有后来对该映射区的引用都是引用该副本，而不是原始文件。（此标志的一种用途是用于调试程序，它将一程序文件的正文部分映射至一存储区，但允许用户修改其中的指令。任何修改只影响程序文件的副本，而不影响原文件。）

每种实现都可能还有另外一些MAP_xxx标志值，它们是这种实现所特有的。详细情况请参见你所使用系统的mmap(2)手册页。

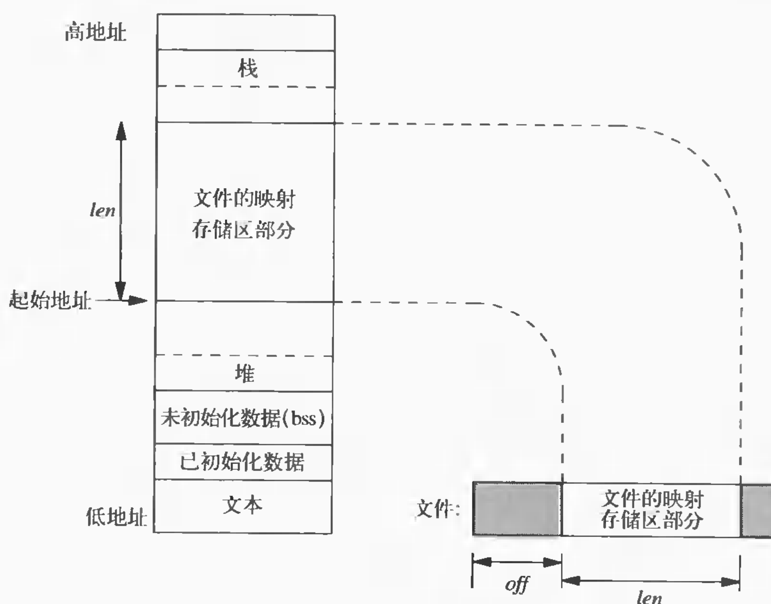


图14-11　存储映射文件的例子

*off*和*addr*的值（如果指定了MAP_FIXED）通常应当是系统虚存页长度的倍数。虚存页长可用带参数_SC_PAGESIZE或_SC_PAGE_SIZE的sysconf函数（见2.5.4节）得到。因为*off*和*addr*常常指定为0，所以这种要求一般并不重要。

因为映射文件的起始偏移量受系统虚存页长度的限制，那么如果映射区的长度不是页长的整数倍时，将如何呢？假定文件长12字节，系统页长为512字节，则系统通常提供512字节的映射区，其中后500字节被设置为0。可以修改这500字节，但任何变动都不会在文件中反映出来。于是，我们不能用mmap将数据添加到文件中。为了做到这一点，我们必须首先加长该文件，这将示于程序清单14-12中。

与映射存储区相关的有SIGSEGV和SIGBUS两个信号。信号SIGSEGV通常用于指示进程试图访问对它不可用的存储区。如果进程企图存数据到mmap指定为只读的映射存储区，那么也产生此信号。如果访问映射区的某个部分，而在访问时这一部分实际上已不存在，则产生SIGBUS信号。例如，用文件长度映射了一个文件，但在引用该映射区之前，另一个进程已将该文件截短，此时，如果进程企图访问对应于该文件已截去部分的映射区，则会接收到SIGBUS信号。

在调用fork之后，子进程继承存储映射区（因为子进程复制父进程地址空间，而存储映射区是该地址空间中的一部分），但是由于同样的理由，调用exec后的新程序则不继承此存储映射区。

调用mprotect可以更改一个现存映射存储区的权限。

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

返回值：若成功则返回0，若出错则返回−1

*prot*的许可值与mmap中*prot*参数一样（表14-9）。地址参数*addr*的值必须是系统页长的整数倍。

> 在Single UNIX Specification中，mprotect函数是存储保护选项中的组成部分，遵循XSI的系统要求支持它。

如果在共享存储映射区中的页已被修改，那么我们可以调用msync将该页冲洗到被映射的文件中。msync函数类似于fsync（3.13节），但作用于存储映射区。

489

```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

返回值：若成功则返回0，若出错则返回−1

如果映射是私有的，那么不修改被映射的文件。与其他存储映射函数一样，地址必须与页边界对齐。

*flags*参数使我们对如何冲洗存储区有某种程度的控制。我们可以指定MS_ASYNC标志以简化被写页的调度。如果我们希望在返回之前等待写操作完成，则可指定MS_SYNC标志。一定要指定MS_ASYNC和MS_SYNC中的一个。

MS_INVALIDATE是一个可选标志，使用它们以通知操作系统丢弃与底层存储器没有同步的任何页。若使用了此标志，某些实现将丢弃在指定范围中的所有页，但这并不是所期望的。

进程终止时，或调用了munmap之后，存储映射区就被自动解除映射。关闭文件描述符*filedes*并不解除映射区。

```
#include <sys/mman.h>

int munmap(caddr_t addr, size_t len);
```

返回值：若成功则返回0，若出错则返回−1

munmap不会影响被映射的对象，也就是说，调用munmap不会使映射区的内容写到磁盘文件上。对于MAP_SHARED区磁盘文件的更新，在写到存储映射区时按内核虚存算法自动进行。在解除了映射后，对于MAP_PRIVATE存储区的修改被丢弃。

实 例

程序清单14-12用存储映射I/O复制一个文件（类似于cp(1)命令）。

程序清单14-12 用存储映射I/O复制文件

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    void        *src, *dst;
    struct stat statbuf;

    if (argc != 3)
        err_quit("usage: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[2]);

    if (fstat(fdin, &statbuf) < 0)  /* need size of input file */
        err_sys("fstat error");

    /* set size of output file */
    if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        err_sys("lseek error");
    if (write(fdout, "", 1) != 1)
        err_sys("write error");

    if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED,
      fdin, 0)) == MAP_FAILED)
        err_sys("mmap error for input");

    if ((dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
      MAP_SHARED, fdout, 0)) == MAP_FAILED)
        err_sys("mmap error for output");

    memcpy(dst, src, statbuf.st_size);  /* does the file copy */
    exit(0);
}
```

490

该程序首先打开两个文件，然后调用fstat得到输入文件的长度。在为输入文件调用mmap和设置输出文件长度时都需使用输入文件长度。调用lseek，然后写一个字节以设置输出文件的长度。如果不设置输出文件的长度，则对输出文件调用mmap也可以，但是对相关存储区的第一次引用会产生SIGBUS。也可使用ftruncate函数来设置输出文件的长度，但是并非所有系统都支持该函数扩充文件长度（见4.13节）。

> 在本书讨论的四种平台上，都可用ftruncate扩展文件。

然后对每个文件调用mmap，将文件映射到存储区，最后调用memcpy将输入缓冲区的内容复制到输出缓冲区。在从输入缓冲区（src）取数据字节时，内核自动读输入文件；在将数据存入输出缓冲区（dst）时，内核自动将数据写到输出文件中。

> 数据被写入文件的确切时间依赖于系统的页管理算法。某些系统设置了守护进程，在系统运行期

间，它"慢条斯理"地将脏页写到磁盘上。如果想要确保数据安全地写到文件中，则需在进程终止前以MS_SYNC标志调用msync。

491

将存储区映射复制与用read，write进行的复制（缓冲区长度为8 192）相比较，得到表14-10中所示的结果。其中，时间单位是秒，被复制文件的长度是300MB。

表14-10　read/write与mmap/memcpy比较的时间结果

操　作	Linux 2.4.22 (Intel x86)			Solaris 9 (SPARC)		
	用户	系统	时钟	用户	系统	时钟
read/write	0.04	1.02	39.76	0.18	9.70	41.66
mmap/memcpy	0.64	1.31	24.26	1.68	7.94	28.53

对于Solaris 9，两种复制方式的CPU时间（用户+系统）几乎相同：9.88s对9.62s。对于Linux 2.4.22，mmap/memcpy方式的CPU时间大约是read/write方式的两倍。这种差别可能是由两种系统实现在处理时间计算方面所使用的方法不同而造成的。

如果考虑到时钟时间，那么mmap和memcpy方式较read和write方式要快。这是合情合理的。使用mmap和memcpy时做的工作要少。用read和write时，要先将数据从内核缓冲区复制到应用程序缓冲区（read），然后又将应用程序缓冲区中的数据复制至内核缓冲区（write）。用mmap和memcpy时，则直接将映射到应用程序地址空间的一个内核缓冲区中的数据复制到另一个同样映射到应用程序地址空间中的内核缓冲区中。□

将一个普通文件复制到另一个普通文件中时，存储映射I/O比较快。但是有一些限制，例如，不能用其在某些设备（例如网络设备或终端设备）之间进行复制，并且在对被复制的文件进行映射后，也要注意该文件的长度是否改变。尽管如此，某些应用程序会从存储映射I/O得到好处，因为它处理的是存储空间而不是读、写一个文件，所以常常可以简化算法。从存储映射I/O中得益的一个例子是对帧缓冲区设备的操作，该设备引用一个位图式显示（bit-mapped display）。

Krieger,Stumm和Unraup[1992]第5章说明了一个使用存储映射I/O的标准I/O库。

15.9节将回过头来讨论存储映射I/O，用一个例子说明如何使用存储映射I/O在有关进程间提供共享存储区。

14.10　小结

本章说明了很多高级I/O功能，其中大多数将在后面章节的例子中使用：

- 非阻塞I/O——发一个I/O操作，不使其阻塞。
- 记录锁（在第20章数据库函数库中有一个实例，将对此作更详细的讨论）。

492

- 系统V流机制（在第17章中，我们将需要使用这部分内容以理解基于STREAMS的管道、传送文件描述符以及系统V的客户/服务器连接）。
- I/O多路转接——select和poll函数（后面的很多实例将用到这两个函数）。
- readv和writev函数（后面的很多实例也将用到这两个函数）。
- 存储映射I/O（mmap）。

习题

14.1 编写一个测试程序以说明你所用系统在下列情况下的运行情况：一个进程在试图对一个文件的某个范围加写锁的时候阻塞，之后其他进程又提出了一些相关的加读锁请求。试图加写锁的进程会不会因其他进程的行为而饿死？

14.2 查看你所用系统的头文件，并研究select和四个FD_宏的实现。

14.3 系统头文件通常对fd_set数据类型可以处理的最大描述符数有一个内置的限制，假设需要将描述符数限制增加到2 048，该如何实现？

14.4 比较处理信号集的函数（见10.11节）和处理fd_set描述符集的函数，并比较在你的系统上实现它们的方法。

14.5 getmsg可以返回多少种不同的信息？

14.6 用select或poll实现一个与sleep类似的函数sleep_us，不同之处是要等待指定的若干微秒。比较这个函数和BSD中的usleep函数。

14.7 是否可以利用建议性记录锁来实现程序清单10-17中的函数TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD？如果可以，编写这些函数并测试其功能。

14.8 用非阻塞写测试管道的容量。将其值与第2章的PIPE_BUF的值比较。

14.9 回忆表14-8，在你的系统上找到一个转折点，从此点开始，使用writev将快于你自己复制数据并使用单个write。

14.10 运行程序清单14-12所列程序复制一个文件，检查输入文件的上一次访问时间是否改变了？

14.11 在程序清单14-12中，在调用mmap后调用close关闭输入文件，以验证关闭描述符不会使内存映射I/O失效。

493

第15章

进程间通信

15.1 引言

第8章说明了进程控制原语并且观察了如何调用多个进程。但是这些进程之间交换信息的方法只能是经由fork或exec传送打开文件，或者通过文件系统。本章将说明进程之间相互通信的其他技术——IPC（InterProcess Communication）。

过去，UNIX系统IPC是各种进程通信方式的统称，但是，其中极少能在所有UNIX系统实现中进行移植。随着POSIX和Open Group（以前是X/Open）标准化的推进和影响的扩大，情况虽已得到改善，但差别仍然存在。表15-1摘要列出了本书讨论的四种实现所支持的不同形式的IPC。

表15-1 UNIX系统IPC摘要

IPC类型	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
半双工管道	•	(全)	•	•	(全)
FIFO	•	•	•	•	•
全双工管道	允许	•,UDS	opt, UDS	UDS	•, UDS
命名全双工管道	XSI 可选	UDS	opt, UDS	UDS	•, UDS
消息队列	XSI	•	•		•
信号量	XSI	•	•	•	•
共享存储	XSI	•	•	•	•
套接字	•	•	•	•	•
STREAMS	XSI 可选		opt		

注意，Single UNIX Specification（“SUS”列）要求的是半双工管道，但允许实现支持全双工管道。若应用程序在编写时假定基础操作系统只支持半双工管道，那么支持全双工管道的实现仍将使这种应用程序正常工作。表中使用“(全)”而非黑点显示用全双工管道支持半双工管道的实现。

在表15-1中的黑点表示基本功能得到支持。对于全双工管道，如果经由UNIX域套接字（见17.3节）支持该特征，则在相应列中标示“UDS”。某些实现用管道和UNIX系统域套接字支持该特征，所以相关位置表示为“UDS”和一个黑点。

正如在14.4节所提到的那样，在Single UNIX Specification中，对STREAMS的支持是可选择的。命名全双工管道是作为已装配的基于STREAMS的管道提供的，所以在Single UNIX Specification中它也是可选的。在Linux中，对STREAMS的支持是可用的，但依靠称为“LiS”（Linux STREAMS）的单独可选择包。在平台对相应特征以可选择包方式提供支持时，相应位

置标示为"opt"，可选择包通常并非是默认安装的。

表15-1中前7种IPC通常限于同一台主机的各个进程间的IPC。最后两种，即套接字和STREAMS，是仅有的两种支持不同主机上各个进程间IPC的类型。

我们将有关IPC的讨论分成3章。本章讨论经典的IPC：管道、FIFO、消息队列、信号量以及共享存储器。下一章将观察使用套接字的网络IPC。第17章将考查IPC的某些高级特征。

15.2　管道

管道是UNIX系统IPC的最古老形式，并且所有UNIX系统都提供此种通信机制。管道有下面两种局限性：

(1) 历史上，它们是半双工的（即数据只能在一个方向上流动）。现在，某些系统提供全双工管道，但是为了最佳的可移植性，我们决不应预先假定系统使用此特性。

(2) 它们只能在具有公共祖先的进程之间使用。通常，一个管道由一个进程创建，然后该进程调用fork，此后父、子进程之间就可应用该管道。

我们将会看到FIFO（见15.5节）没有第二种局限性，UNIX域套接字（见17.3）和命名流管道（见17.2.2节）则没有这两种局限性。

尽管有这两种局限性，半双工管道仍是最常用的IPC形式。　每当你在管道线中键入一个由shell执行的命令序列时，shell为每一条命令单独创建一进程，然后将前一条命令进程的标准输出用管道与后一条命令的标准输入相连接。

496

管道是由调用pipe函数而创建的：

```
#include <unistd.h>

int pipe(int filedes[2]);
```

返回值：若成功则返回0，若出错则返回−1

经由参数*filedes*返回两个文件描述符：*filedes[0]*为读而打开，*filedes[1]*为写而打开。*filedes[1]*的输出是*filedes[0]*的输入。

> 在4.3BSD、4.4BSD和Mac OS X 10.3中，管道是用UNIX域套接字实现的。虽然UNIX域套接字是默认全双工的，但这些操作系统对用于管道的套接字进行了处理，使这些管道只以半双工模式操作。
>
> POSIX.1允许实现支持全双工管道。对于这些实现，*filedes[0]*和*filedes[1]*以读/写方式打开。

有两种方法来描绘一个半双工管道，见图15-1。左半图显示了管道的两端在一个进程中相互连接，右半图则说明数据通过内核在管道中流动。

fstat函数（见4.2节）对管道的每一端都返回一个FIFO类型的文件描述符，可以用S_ISFIFO宏来测试管道。

> POSIX.1规定stat结构的st_size成员对于管道是未定义的。但是当fstat函数应用于管道读端的文件描述符时，很多系统在st_size中存放管道中可用于读的字节数。但是，这是不可移植的。

单个进程中的管道几乎没有任何用处。通常，调用pipe的进程接着调用fork，这样就创建了从父进程到子进程（或反向）的IPC通道。图15-2显示了这种情况。

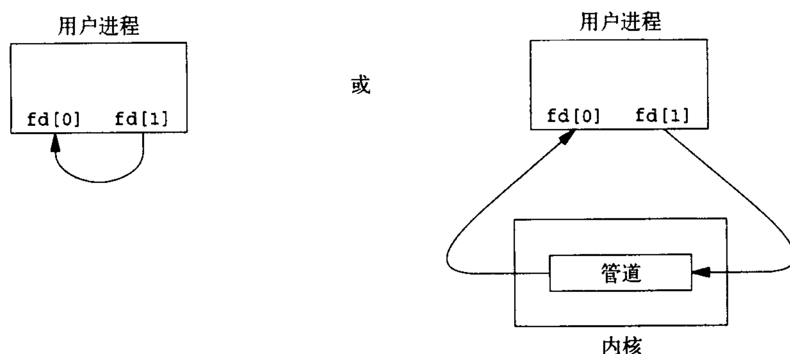


图15-1　观察半双工管道的两种方法

497

调用fork之后做什么取决于我们想要有的数据流的方向。对于从父进程到子进程的管道，父进程关闭管道的读端（fd[0]），子进程则关闭写端（fd[1]）。图15-3显示了在此之后描述符的安排。

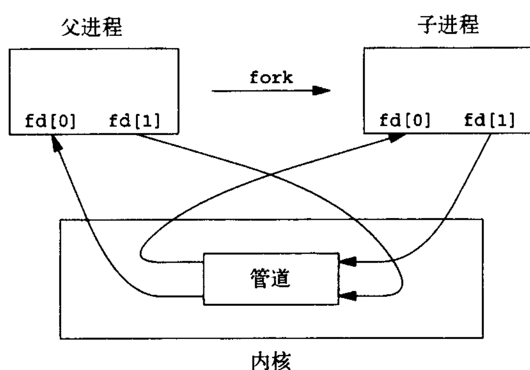


图15-2　调用fork之后的半双工管道

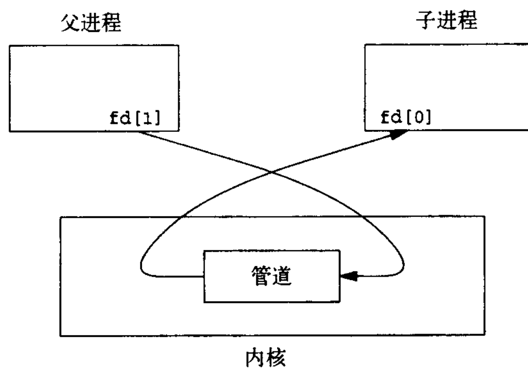


图15-3　从父进程到子进程的管道

为了构造从子进程到父进程的管道，父进程关闭fd[1]，子进程关闭fd[0]。

当管道的一端被关闭后，下列两条规则起作用：

(1) 当读一个写端已被关闭的管道时，在所有数据都被读取后，read返回0，以指示达到了文件结束处。（从技术方面考虑，管道的写端还有进程时，就不会产生文件的结束。可以复制一个管道的描述符，使得有多个进程对它具有写打开文件描述符。但是，通常一个管道只有一个读进程、一个写进程。下一节介绍FIFO时，我们会看到对于一个单一的FIFO常常有多个写进程。）

498

(2) 如果写一个读端已被关闭的管道，则产生信号SIGPIPE。如果忽略该信号或者捕捉该信号并从其处理程序返回，则write返回-1，errno设置为EPIPE。

在写管道（或FIFO）时，常量PIPE_BUF规定了内核中管道缓冲区的大小。如果对管道调用write，而且要求写的字节数小于等于PIPE_BUF，则此操作不会与其他进程对同一管道（或FIFO）的write操作穿插进行。但是，若有多个进程同时写一个管道（或FIFO），而且有进程要求写的字节数超过PIPE_BUF字节数时，则写操作的数据可能相互穿插。用pathconf或fpathconf函数（见表2-11）可以确定PIPE_BUF的值。

实例

程序清单15-1创建了一个从父进程到子进程的管道，并且父进程经由该管道向子进程传送

数据。

程序清单15-1 经由管道父进程向子进程传送数据

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

□

在上面的例子中，直接对管道描述符调用read和write。更好的方法是将管道描述符复制为标准输入和标准输出。在此之后通常子进程执行另一个程序，该程序或者从标准输入（已创建的管道）读数据，或者将数据写至其标准输出（该管道）。

499

试编写一个程序，其功能是每次一页显示已产生的输出。已经有很多UNIX系统实用程序具有分页功能，因此无需再构造一个新的分页程序，而是调用用户最喜爱的分页程序。为了避免先将所有数据写到一个临时文件中，然后再调用系统中有关程序显示该文件，我们希望将输出通过管道直接送到分页程序。为此，先创建一个管道，调用fork产生一个子进程，使子进程的标准输入成为管道的读端，然后调用exec，执行用户喜爱的分页程序。程序清单15-2显示了如何实现这些操作。（本例要求在命令行中有一个参数说明要显示文件的名称。通常，这种类型的程序要求在终端上显示的数据已经在存储器中。）

程序清单15-2 将文件复制到分页程序

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER   "/bin/more"     /* default pager program */

int
main(int argc, char *argv[])
{
    int     n;
    int     fd[2];
    pid_t   pid;
```

```
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                           /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");

        exit(0);
    } else {                                        /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
    exit(0);
}
```

500

在调用fork之前先创建一个管道。fork之后父进程关闭其读端，子进程关闭其写端。子进程然后调用dup2，使其标准输入成为管道的读端。当执行分页程序时，其标准输入将是管道的读端。

当我们将一个描述符复制到另一个时（在子进程中，fd[0]复制到标准输入），应当注意在复制之前该描述符的值并不是所希望的值。如果该描述符已经具有所希望的值，并且我们先调用dup2，然后调用close则将关闭此进程中只有该单个描述符所代表的打开文件。（回忆3.12节中所述，当dup2中的两个参数值相等时的操作。）在本程序中，如果shell没有打开标准

输入，那么程序开始处的fopen应已使用描述符0，也就是最小未使用的描述符，所以fd[0]决不会等于标准输入。尽管如此，只要先调用dup2，然后调用close以复制一个描述符到另一个，作为一种保护性的编程措施，我们总是先将两个描述符进行比较。

请注意，我们是如何使用环境变量PAGER试图获得用户分页程序名称的。如果这种操作没有成功，则使用系统默认值。这是环境变量的常见用法。 □

回忆8.9节中的5个函数：TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD。程序清单10-17提供了一个使用信号的实现。程序清单15-3则是一个使用管道的实现。

501

程序清单15-3 使父、子进程同步的例程

```
#include "apue.h"

static int  pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
```

```
        err_sys("read error");

    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

502

如图15-4所示，在fork之前创建了两个管道。父进程在调用TELL_CHILD时，写一个字符“p”至上一个管道，子进程在调用TELL_PARENT时，经由下一个管道写一个字符“c”。相应的WAIT_xxx函数调用read读这个字符，并发生阻塞。

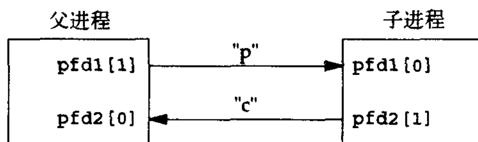


图15-4 用两个管道实现父子进程同步

请注意，每一个管道都有一个额外的读取进程，这没有关系。也就是说，除了子进程从pfd1[0]读取，父进程也有上一个管道的读端。因为父进程并没有执行对该管道的读操作，所以这不会产生任何影响。 □

15.3 popen和pclose函数

常见的操作是创建一个管道连接到另一个进程，然后读其输出或向其输入端发送数据，为此，标准I/O库提供了两个函数popen和pclose。这两个函数实现的操作是：创建一个管道，调用fork产生一个子进程，关闭管道的不使用端，执行一个shell以运行命令，然后等待命令终止。

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

返回值：若成功则返回文件指针，若出错则返回NULL

```
int pclose(FILE *fp);
```

返回值：*cmdstring*的终止状态，若出错则返回−1

函数popen先执行fork，然后调用exec以执行*cmdstring*，并且返回一个标准I/O文件指针。如果*type*是"r"，则文件指针连接到*cmdstring*的标准输出（见图15-5）。

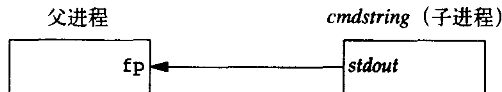


图15-5 执行fp = popen(*cmdstring*, "r")函数的结果

503

如果*type*是"w"，则文件指针连接到*cmdstring*的标准输入（见图15-6）。



图15-6 执行fp = popen(*cmdstring*, "w")函数的结果

有一种方法可以帮助我们记住popen最后一个参数及其作用，这就是与fopen进行类比。如果

*type*是“r”,则返回的文件指针是可读的，如果*type*是“w”，则是可写的。

pclose函数关闭标准I/O流，等待命令执行结束，然后返回shell的终止状态。(我们曾在8.6节对终止状态进行过说明，system函数（见8.13节）也返回终止状态。）如果shell不能被执行，则pclose返回的终止状态与shell已执行exit(127)一样。

*cmdstring*由Bourne shell以下列方式执行：

sh -c *cmdstring*

这表示shell将扩展*cmdstring*中的任何特殊字符。例如，可以使用：

```
fp = popen("ls *.c", "r");
```

或者

```
fp = popen("cmd 2>&1", "r");
```

用popen重写程序清单15-2，其结果示于程序清单15-4中。

程序清单15-4　用popen向分页程序传送文件

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER   "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE    *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    /* copy argv[1] to pager */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");

    exit(0);
}
```

504

使用popen减少了需要编写的代码量。

shell命令${PAGER:-more}的意思是：如果shell变量PAGER已经定义，且其值非空，则使用其值，否则使用字符串more。 □

实例：popen和pclose函数

程序清单15-5是我们编写的popen和pclose版本。

程序清单15-5 popen和pclose函数

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Pointer to array allocated at run-time.
 */
static pid_t    *childpid = NULL;

/*
 * From our open_max(), Figure 2.16.
 */
static int      maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int     i;
    int     pfd[2];
    pid_t   pid;
    FILE    *fp;

    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;     /* required by POSIX */
        return(NULL);
    }

    if (childpid == NULL) {     /* first time through */
        /* allocate zeroed out array for child pids */
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL);   /* errno set by pipe() */

    if ((pid = fork()) < 0) {
        return(NULL);   /* errno set by fork() */
    } else if (pid == 0) {                              /* child */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }
```

505

```
        /* close all descriptors in childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }

    /* parent continues... */
    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}
int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

506

虽然popen的核心部分与本章中以前用过的代码类似，但是增加了很多需要考虑的细节。首先，每次调用popen时，应当记住所创建的子进程的进程ID，以及其文件描述符或FILE指针。我们选择在数组childpid中保存子进程ID，并用文件描述符作为其下标。于是，当以FILE指针作为参数调用pclose时，我们调用标准I/O函数fileno得到文件描述符，然后取得子进程ID，并用其作为参数调用waitpid。因为一个进程可能调用popen多次，所以在动态分配childpid数组时（第一次调用popen时），其数组长度应当是最大文件描述符数，于是该

数组中可以存放与最大文件描述符数相同的子进程。

调用pipe、fork以及为每个进程复制相应的文件描述符，这些操作与本章前面所述的类似。

POSIX.1要求子进程关闭在以前调用popen时打开且当前仍旧打开的所有I/O流。为此，在子进程中从头逐个检查childpid数组的各元素，关闭仍旧打开的任何描述符。

若pclose的调用者已经为信号SIGCHLD设置了一个信号处理程序，则pclose中的waitpid调用将返回一个EINTR。因为允许调用者捕捉此信号（或者任何其他可能中断waitpid调用的信号），所以当waitpid被一个捕捉到的信号中断时，我们只是再次调用waitpid。

507

注意，如果应用程序调用waitpid，并且获得popen所创建的子进程的终止状态，则在应用程序调用pclose时，其中将调用waitpid，它发现子进程已不再存在，此时返回-1，errno则被设置为ECHILD。这正是POSIX.1所要求的。

> 如果一个信号中断了wait，pclose的早期版本返回EINTR。pclose的早期版本在wait期间阻塞或忽略信号SIGINT、SIGQUIT以及SIGHUP。POSIX.1则不允许这一点。

□

注意，popen决不应由设置用户ID或设置组ID程序调用。当它执行命令时，popen等同于：

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

它在从调用者继承的环境中执行shell，并由shell解释执行*command*。一个心怀不轨的用户可以操纵这种环境，使得shell能以设置ID文件模式所授与的提升了的权限以及非预期的方式执行命令。

popen特别适用于构造简单的过滤器程序，它变换运行命令的输入或输出。当命令希望构造它自己的管道线时，就是这种情形。

实 例

考虑一个应用程序，它向标准输出写一个提示，然后从标准输入读1行。使用popen，可以在应用程序和输入之间插入一个程序以便对输入进行变换处理。图15-7显示了为此做的进程安排。

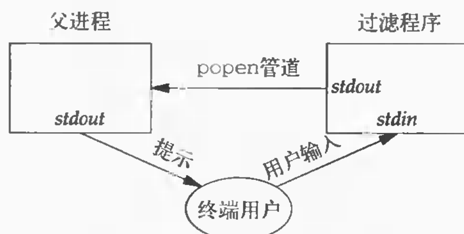


图15-7 用popen对输入进行变换处理

对输入进行的变换可能是路径名扩充，或者是提供一种历史机制（记住以前输入的命令）。

程序清单15-6是一个简单的过滤程序，它只是将标准输入复制到标准输出，在复制时将所有大写字符变换为小写字符。在写了一行之后，对标准输出进行了冲洗（用fflush），其理由将在下一节介绍协同进程时讨论。

508

程序清单15-6　将大写字符换成小写字符的过滤程序

```
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

对该过滤程序进行编译，其可执行目标代码存放在文件myuclc中，然后在程序清单15-7中用popen调用它们。

程序清单15-7　调用大写/小写过滤程序以读取命令

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;

    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

509

因为标准输出通常是行缓冲的，而提示并不包含换行符，所以在写了提示之后，需要调用fflush。 □

15.4　协同进程

UNIX系统过滤程序从标准输入读取数据，对其进行适当处理后写到标准输出。几个过滤

程序通常在shell管道命令行中线性地连接。当一个程序产生某个过滤程序的输入，同时又读取该过滤程序的输出时，则该过滤程序就成为协同进程（coprocess）。

Korn shell提供了协同进程[Bolsky and Korn 1995]。Bourne shell、Bourne-again shell和C shell并没有提供按协同进程方式将进程连接起来的方法。协同进程通常在shell的后台运行，其标准输入和标准输出通过管道连接到另一个程序。虽然初始化一个协同进程并将其输入和输出连接到另一个进程，用到的shell语法是十分奇特的（详细情况见Bolsky和Korn[1995]中的第62~63页），但是协同进程的工作方式在C程序中也是非常有用的。

popen只提供连接到另一个进程的标准输入或标准输出的一个单向管道，而对于协同进程，则它有连接到另一个进程的两个单向管道——一个接到其标准输入，另一个则来自其标准输出。我们先要将数据写到其标准输入，经其处理后，再从其标准输出读取数据。



让我们通过一个实例来观察协同进程。进程先创建两个管道：一个是协同进程的标准输入，另一个是协同进程的标准输出。图15-8显示了这种安排。

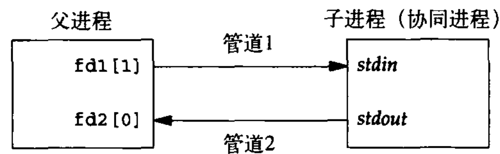


图15-8 写协同进程的标准输入，读它的标准输出

程序清单15-8程序是一个简单的协同进程，它从其标准输入读两个数，计算它们的和，然后将结果写至标准输出。（协同进程通常会做较此更有意义的工作。设计本实例的目的是帮助了解将进程连接起来所需的各种管道设施。）

510

程序清单15-8 对两个数求和的简单过滤程序

```
#include "apue.h"

int
main(void)
{
    int     n, int1, int2;
    char    line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

对此程序进行编译，将其可执行目标代码存入名为add2的文件。

程序清单15-9从其标准输入读入两个数之后调用add2协同进程，并将协同进程送来的值写到其标准输出。

程序清单15-9 驱动add2过滤程序的程序

```
#include "apue.h"

static void sig_pipe(int);      /* our signal handler */

int
main(void)
{
    int     n, fd1[2], fd2[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                           /* parent */
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }

        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                    /* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd1[0]);
        }

        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
```

511

```
            err_sys("execl error");
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

在程序中创建了两个管道，父、子进程各自关闭它们不需使用的端口。两个管道一个用做协同进程的标准输入，另一个则用做它的标准输出。子进程调用dup2使管道描述符移至其标准输入和标准输出，然后调用execl。

512

若编译和运行程序清单15-9程序，它如所希望的那样进行工作。进而言之，在程序正等待输入时，若先杀死add2协同进程，然后输入两个数，接着程序对管道进行写操作，此时，由于该管道已无读进程，于是调用信号处理程序（见习题15.4）。

回忆表15-1，并非所有系统用pipe函数提供全双工管道。程序清单17-1将提供这一实例的另一个版本，它使用一个全双工管道而不是两个半双工管道，适用于支持全双工管道的系统。 □

实例

在协同进程add2（见程序清单15-8）中，有意地使用了read和writeI/O（UNIX系统调用）。如果使用标准I/O改写该协同进程，其后果是什么呢？程序清单15-10就是改写后的版本。

程序清单15-10 对两个数求和的滤波程序，使用标准I/O

```
#include "apue.h"

int
main(void)
{
    int     int1, int2;
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("printf error");
        } else {
            if (printf("invalid args\n") == EOF)
                err_sys("printf error");
        }
    }
    exit(0);
}
```

若程序清单15-9调用此新的协同进程，则它不再工作。问题出在系统默认的标准I/O缓冲机制上。当调用程序清单15-10所示程序时，对标准输入的第一个fgets引起标准I/O库分配一个缓冲区，并选择缓冲区的类型。因为标准输入是个管道，所以标准I/O库由系统默认是全缓冲的。

对标准输出也作同样的处理。当add2从其标准输入读取而发生阻塞时，程序清单15-9程序从管道读时也发生阻塞，于是产生了死锁。

为此，更改将要运行的协同进程的缓冲类型，在程序清单15-10中的while循环之前加上下面4行：

513

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
```

这些代码行使得当有一行可用时，fgets就返回，并使得当输出一换行符时，printf立即执行fflush操作。对setvbuf进行的这些显式调用使得程序15-10能正常工作。

如果不能修改这种协同进程程序，则需使用其他技术。例如，如果在程序中使用awk(1)代替add2作为协同进程，则下列命令行不能工作：

```
#! /bin/awk -f
{ print $1 + $2 }
```

不能工作的原因还是标准I/O的缓冲机制问题。但是，在这种情况下不能改变awk的工作方式（除非有awk的源代码）。我们不能修改awk的可执行代码，于是也就不能更改处理其标准I/O缓冲的方式。

对这种问题的一般解决方法是使被调用的协同进程（在本例中是awk）认为它的标准输入和输出都被连接到一个终端。这使得协同进程中的标准I/O例程对这两个I/O流进行行缓冲，这类似于前面所做的显式setvbuf调用。第19章将用伪终端实现这一点。□

15.5　FIFO

FIFO有时被称为命名管道。管道只能由相关进程使用，这些相关进程的共同的祖先进程创建了管道。（一个例外是已装配的基于STREAMS的管道，我们将在17.2.2中对此进行说明。）但是，通过FIFO，不相关的进程也能交换数据。

第4章中已经提及FIFO是一种文件类型。stat结构（见4.2节）成员st_mode的编码指明文件是否是FIFO类型。可以用S_ISFIFO宏对此进行测试。

创建FIFO类似于创建文件。确实，FIFO的路径名存在于文件系统中。

```
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

返回值：若成功则返回0，若出错则返回−1

mkfifo函数中*mode*参数的规格说明与open函数中的*mode*相同（见3.3节）。新FIFO的用户和组的所有权规则与4.6节所述的相同。

514

一旦已经用mkfifo创建了一个FIFO，就可用open打开它。其实，一般的文件I/O函数（close、read、write、unlink等）都可用于FIFO。

> 应用程序可以用mknod函数创建FIFO。POSIX.1原先并没有包括mknod函数，它首先提出了mkfifo。mknod现在已包括在XSI扩展中。在大多数系统中，mkfifo调用mknod创建FIFO。

> POSIX.1也包括了对mkfifo(1)命令的支持。本书讨论的四种平台都支持此命令。于是，用一条shell命令就可以创建一个FIFO，然后用一般的shell I/O重定向对其进行访问。

当打开一个FIFO时，非阻塞标志（O_NONBLOCK）产生下列影响：

- 在一般情况中（没有指定O_NONBLOCK），只读open要阻塞到某个其他进程为写而打开此FIFO。类似地，只写open要阻塞到某个其他进程为读而打开它。
- 如果指定了O_NONBLOCK，则只读open立即返回。但是，如果没有进程已经为读而打开一个FIFO，那么只写open将出错返回-1，其errno是ENXIO。

类似于管道，若用write写一个尚无进程为读而打开的FIFO，则产生信号SIGPIPE。若某个FIFO的最后一个写进程关闭了该FIFO，则将为该FIFO的读进程产生一个文件结束标志。

一个给定的FIFO有多个写进程是很常见的。这就意味着如果不希望多个进程所写的数据互相穿插，则需考虑原子写操作。（在17.2.2节中将说明解决此问题的一种方法。）正如对于管道一样，常量PIPE_BUF说明了可被原子地写到FIFO的最大数据量。

FIFO有下面两种用途：

(1) FIFO由shell命令使用以便将数据从一条管道线传送到另一条，为此无需创建中间临时文件。

(2) FIFO用于客户进程–服务器进程应用程序中，以在客户进程和服务器进程之间传递数据。我们各用一个例子来说明这两种用途。

实例：用FIFO复制输出流

FIFO可被用于复制串行管道命令之间的输出流，于是也就不需要写数据到中间磁盘文件中（类似于使用管道以避免中间磁盘文件）。管道只能用于进程间的线性连接，然而，因为FIFO具有名字，所以它可用于非线性连接。

考虑这样一个操作过程，它需要对一个经过过滤的输入流进行两次处理。图15-9表示了这种安排。

515

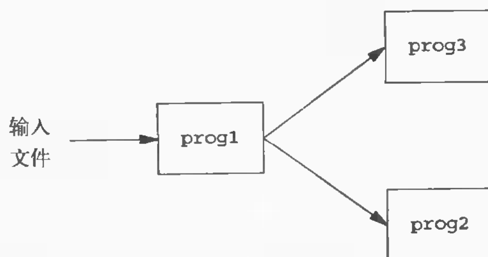


图15-9 对一个经过过滤的输入流进行两次处理

使用FIFO以及UNIX系统程序tee(1)，就可以实现这样的过程而无需使用临时文件。（tee程序将其标准输入同时复制到其标准输出以及其命令行中包含的命名文件中。）

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

我们创建FIFO，然后在后台启动prog3，它从FIFO读数据。然后启动prog1，用tee将其输出发送到FIFO和prog2。图15-10显示了有关安排。

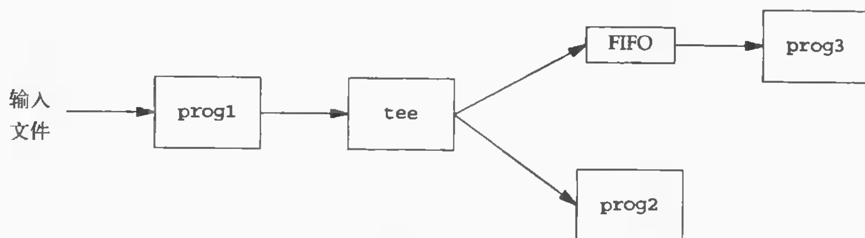


图15-10　使用FIFO和tee将一个流发送到两个进程

实例：客户进程-服务器进程使用FIFO进行通信

FIFO的另一个应用是在客户进程和服务器进程之间传送数据。如果有一个服务器进程，它与很多客户进程有关，则每个客户进程都可将其请求写到一个该服务器进程创建的众所周知的FIFO中（"众所周知"的意思是：所有需与服务器进程联系的客户进程都知道该FIFO的路径名）。图15-11显示了这种安排。因为对于该FIFO有多个写进程，客户进程发送给服务器进程的请求其长度要小于PIPE_BUF字节。这样就能避免客户多个write之间的交错。

516

在这种类型的客户进程-服务器进程通信中使用FIFO的问题是：服务器进程如何将回答送回各个客户进程。不能使用单个FIFO，因为服务器进程会发出对各个客户进程请求的响应，而请求者却不可能知道什么时候去读才能恰如其分地读到对它的响应。一种解决方法是每个客户进程都在其请求中包含它的进程ID。然后服务器进程为每个客户进程创建一个FIFO，所使用的路径名是以客户进程的进程ID为基础的。例如，服务器进程可以用名字/tmp/serv1.XXXXX创建FIFO，其中XXXXX被替换成客户进程的进程ID。图15-12显示了这种安排。

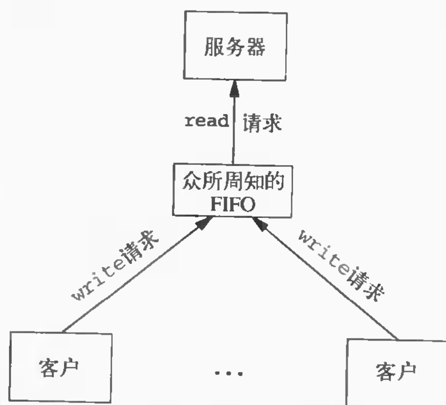


图15-11　客户进程用FIFO向服务器进程发送请求

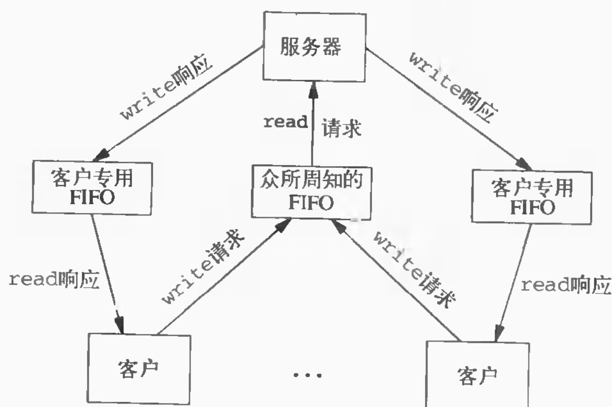


图15-12　客户进程-服务器进程用FIFO进行通信

517

这种安排可以工作，但也有一些不足之处。其中之一是服务器进程不能判断一个客户进程是否崩溃终止，这就使得客户进程专用的FIFO会遗留在文件系统中。另一个不足之处是服务器进程必须捕捉SIGPIPE信号，因为客户进程在发送一个请求后没有读取响应就可能终止，于是留下一个只有写进程（服务器进程）而无读进程的客户进程专用FIFO。在17.2.2节中我们将讨论已装配的基于STREAMS的管道以及connld，那时会说明解决此种问题更加妥善的方法。

按照图15-12中的安排，如果服务器进程以只读方式打开众所周知的FIFO（因为它只需读

该FIFO），则每当客户进程数从1变成0时，服务器进程就将在FIFO中读到一个文件结束标记。为使服务器进程免于处理这种情况，一种常用的技巧是使服务器进程以读-写方式打开其FIFO（见习题15.10）。 □

15.6 XSI IPC

有三种IPC我们称作XSI IPC，即消息队列、信号量以及共享存储器，它们之间有很多相似之处。以下各节将说明这些IPC的各自特殊功能，本节先介绍它们相类似的特征。

XSI IPC源自系统V的IPC功能，二者密切相关。后者源自于1970年的一种称为Columbus UNIX的AT&T内部版本，后来它们被加到系统V上。由于XSI IPC不使用文件系统名字空间，而是构造了它们自己的名字空间，为此常常受到批评。

回忆表15-1，消息队列、信号量及共享存储器定义在Single UNIX Specification的XSI扩展中。

15.6.1 标识符和键

每个内核中的IPC结构（消息队列、信号量或共享存储段）都用一个非负整数的标识符（identifier）加以引用。例如，为了对一个消息队列发送或取消息，只需要知道其队列标识符。与文件描述符不同，IPC标识符不是小的整数。当一个IPC结构被创建，以后又被删除时，与这种结构相关的标识符连续加1，直至达到一个整型数的最大正值，然后又回转到0。

标识符是IPC对象的内部名。为使多个合作进程能够在同一IPC对象上会合，需要提供一个外部名方案。为此使用了键（key），每个IPC对象都与一个键相关联，于是键就用作为该对象的外部名。

无论何时创建IPC结构（调用msgget、semget或shmget），都应指定一个键，键的数据类型是基本系统数据类型key_t，通常在头文件<sys/types.h>中被定义为长整型。键由内核变换成标识符。

有多种方法使客户进程和服务器进程在同一IPC结构上会合： 518

(1) 服务器进程可以指定键IPC_PRIVATE创建一个新IPC结构，将返回的标识符存放在某处（例如一个文件）以便客户进程取用。键IPC_PRIVATE保证服务器进程创建一个新IPC结构。这种技术的缺点是：服务器进程要将整型标识符写到文件中，此后客户进程又要读文件取得此标识符。

IPC_PRIVATE键也可用于父、子进程关系。父进程指定IPC_PRIVATE创建一个新IPC结构，所返回的标识符在调用fork后可由子进程使用。接着，子进程又可将此标识符作为exec函数的一个参数传给一个新程序。

(2) 在一个公用头文件中定义一个客户进程和服务器进程都认可的键。然后服务器进程指定此键创建一个新的IPC结构。这种方法的问题是该键可能已与一个IPC结构相结合，在此情况下，get函数（msgget、semget或shmget）出错返回。服务器进程必须处理这一错误，删除已存在的IPC结构，然后试着再创建它。

(3) 客户进程和服务器进程认同一个路径名和项目ID（项目ID是0~255之间的字符值），接着调用函数ftok将这两个值变换为一个键。然后在方法(2)中使用此键。ftok提供的唯一服务就是由一个路径名和项目ID产生一个键。

```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

返回值：若成功则返回键，若出错则返回(key_t)-1

*path*参数必须引用一个现存文件。当产生键时，只使用*id*参数的低8位。

ftok创建的键通常是用下列方式构成的：按给定的路径名取得其stat结构（见4.2节），从该结构中取出部分st_dev和st_ino字段，然后再与项目ID组合起来。如果两个路径名引用两个不同的文件，那么，对这两个路径名调用ftok通常返回不同的键。但是，因为i节点号和键通常都存放在长整型中，于是创建键时可能会丢失信息。这意味着，如果使用同一项目ID，那么对于不同文件的两个路径名可能产生相同的键。

三个get函数（msgget、semget和shmget）都有两个类似的参数：一个*key*和一个整型*flag*。如若满足下列两个条件之一，则创建一个新的IPC结构（通常由服务器进程创建）：

- *key*是IPC_PRIVATE；
- *key*当前未与特定类型的IPC结构相结合，并且*flag*中指定了IPC_CREAT位。

为访问现存的队列（通常由客户进程进行），*key*必须等于创建该队列时所指定的键，并且不应指定IPC_CREAT。

注意，为了访问一个现存队列，决不能指定IPC_PRIVATE作为键。因为这是一个特殊的键值，它总是用于创建一个新队列。为了访问一个用IPC_PRIVATE键创建的现存队列，一定要知道与该队列相结合的标识符，然后在其他IPC调用中（例如msgsnd和msgrcv）使用该标识符。

519

如果希望创建一个新的IPC结构，而且要确保不是引用具有同一标识符的一个现行IPC结构，那么必须在*flag*中同时指定IPC_CREAT和IPC_EXCL位。这样做了以后，如果IPC结构已经存在就会造成出错，返回EEXIST（这与指定了O_CREAT和O_EXCL标志的open相类似）。

15.6.2 权限结构

XSI IPC为每一个IPC结构设置了一个ipc_perm结构。该结构规定了权限和所有者。它至少包括下列成员：

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user id */
  gid_t  gid;  /* owner's effective group id */
  uid_t  cuid; /* creator's effective user id */
  gid_t  cgid; /* creator's effective group id */
  mode_t mode; /* access modes */
   .
   .
   .
};
```

每种实现在其ipc_perm结构中会包括另外一些成员。如欲了解你所用系统中它的完整定义，请参见<sys/ipc.h>。

在创建IPC结构时，对所有字段都赋初值。以后，可以调用msgctl、semctl或shmctl修改uid、gid和mode字段。为了改变这些值，调用进程必须是IPC结构的创建者或超级用户。更改这些字段类似于对文件调用chown和chmod。

mode字段的值类似于表4-5中所示的值，但是对于任何IPC结构都不存在执行权限。另外，

消息队列和共享存储使用术语读（read）和写（write），而信号量则用术语读（rend）和更改（alter）。表15-2中对每种IPC说明了6种权限。

表15-2 XSI IPC 权限

权 限	位
用户读 用户写（更改）	0400 0200
组读 组写（更改）	0040 0020
其他读 其他写（更新）	0004 0002

某些实现定义了表示每种权限的符号常量，但是这些常量并不包括在Single UNIX Specification中。

520

15.6.3 结构限制

三种形式的XSI IPC都有内置限制（built-in limit）。这些限制的大多数可以通过重新配置内核而加以更改。当叙说每种IPC时，我们都会指出它的限制。

在报告和修改限制方面，每种平台都提供它自己的方法。FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3提供了`sysctl`命令，用该命令观察和修改内核配置参数。Solaris9修改内核配置参数的方法是，修改文件`/etc/system`，然后重启动。

在Linux中，你可以运行`ipcs -l`以显示IPC相关的限制。在FreeBSD中，等效的命令是`ipcs -T`。在Solaris中，运行`sysdef -i`则可找到可调节参数。

15.6.4 优点和缺点

XSI IPC的主要问题是：IPC结构是在系统范围内起作用的，没有访问计数。例如，如果进程创建了一个消息队列，在该队列中放入了几则消息，然后终止，但是该消息队列及其内容并不会被删除。它们余留在系统中直至出现下述情况：由某个进程调用`msgrcv`或`msgctl`读消息或删除消息队列；或某个进程执行`ipcrm`(1)命令删除消息队列；或由正在再启动的系统删除消息队列。将此与管道相比，当最后一个访问管道的进程终止时，管道就被完全地删除了。对于FIFO而言，虽然当最后一个引用FIFO的进程终止时其名字仍保留在系统中，直至显式地删除它，但是留在FIFO中的数据却在此时全部被删除，于是也就徒有其名了。

XSI IPC的另一个问题是：这些IPC结构在文件系统中没有名字。我们不能用第3、第4章中所述的函数来访问它们或修改它们的特性。为了支持它们不得不增加了十几条全新的系统调用（`msgget`、`semop`、`shmat`等）。我们不能用`ls`命令见到IPC对象，不能用`rm`命令删除它们，也不能用`chmod`命令更改它们的访问权限。于是，就不得不增加新的命令`ipcs`(1)和`ipcrm`(1)。

因为这些IPC不使用文件描述符，所以不能对它们使用多路转接I/O函数：`select`和`poll`。这就使得难于一次使用多个IPC结构，以及在文件或设备I/O中使用IPC结构。例如，没有某种形式的忙-等待循环，就不能使一个服务器进程等待将要放在两个消息队列任一个中的消息。

Andrade、Carges和Kovach[1989]对使用系统V IPC的一个事务处理系统进行了综述。他们认为系统V IPC使用的名字空间（标识符）是一个优点而不是前面所说的问题，理由是使用标识符使一个进程只要使用单个函数调用（msgsnd）就能将一个消息发送到一个消息队列，而其他形式的IPC则通常要求open、write和close。这种论点是错误的。为了避免使用键和调用msgget，客户进程总要以某种方式获得服务器进程队列的标识符。分派给特定队列的标识符，取决于在创建该队列时有多少消息队列已经存在，也取决于自内核自举以来，内核中将分配给新队列的表项已经使用了多少次。这是一个动态值，不能被猜出或事先存放在一个头文件中。正如15.6.1节所述，至少服务器进程应将分配给队列的标识符写到一个文件中以便客户进程读取。

521

这些作者列举的消息队列的其他优点是：(a) 可靠，(b) 流是受控的，(c) 面向记录，(d) 可以用非先进先出方式处理。正如在14.4节中所见，STREAMS也具有所有这些优点，两者之间的差别是，在向流发送数据之前需要一个open，在结束时需要一个close。表15-3对这些不同形式IPC的某些特征进行了比较。

表15-3 不同形式IPC之间的特征比较

IPC 类型	无连接?	可靠?	流控制?	记录?	消息类型或优先级?
消息队列	否	是	是	是	是
STREAMS	否	是	是	是	是
UNIX域流套接字	否	是	是	否	否
UNIX域数据报套接字	是	是	否	是	否
FIFO（非STREAMS）	否	是	是	否	否

（第16章将对UNIX流和数据报套接字进行说明。17.3节将说明UNIX域套接字。）表15-3中的“无连接”指的是无需先调用某种形式的打开函数就能发送消息的能力。正如前述，因为需要有某种技术以获得队列标识符，所以我们并不认为消息队列具有无连接特性。因为所有这些形式的IPC都限制用在单主机上，所以它们都是可靠的。当消息通过网络传送时，丢失消息的可能性就要加以考虑。“流控制”指的是：如果系统资源（缓冲区）短缺或者如果接收进程不能再接收更多消息，则发送进程就要休眠。当流控制条件消失时，发送进程应自动地被唤醒。

表15-3中没有表示的一个特征是：IPC设施能否自动地为每个客户进程创建一个到服务器进程的唯一连接。第17章将说明，STREAMS以及UNIX流套接字可以提供这种能力。

下面三节顺次对三种形式的XSI IPC进行详细说明。

15.7 消息队列

消息队列是消息的链接表,存放在内核中并由消息队列标识符标识。在本节中，我们把消息队列简称为队列（queue），其标识符为队列ID（queue ID）。

> Single UNIX Specification在其实时扩展的消息传送选项中包括一种替代的IPC消息队列。本书不讨论实时扩展。

522

msgget用于创建一个新队列或打开一个现存的队列。msgsnd将新消息添加到队列尾端。每个消息包含一个正长整型类型字段，一个非负长度以及实际数据字节（对应于长度），所有

这些都在将消息添加到队列时，传送给msgsnd。msgrcv用于从队列中取消息。我们并不一定要以先进先出次序取消息，也可以按消息的类型字段取消息。

每个队列都有一个msqid_ds结构与其相关联：

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
  msgqnum_t        msg_qnum;     /* # of messages on queue */
  msglen_t         msg_qbytes;   /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */
    .
    .
    .
};
```

此结构规定了队列的当前状态。结构中所示的各成员是由Single UNIX Specification定义的。具体实现可能包括标准中没有定义的另一些字段。

表15-4列出了影响消息队列的系统限制。表中"notsup"表示相关平台不支持该特征；"derived"表示这种限制是从其他限制导出的。例如，在Linux系统中，消息最大数基于队列最大数值和队列中允许数据量的最大值。如果最短消息长度是1字节，则系统范围内的消息数限制是最大消息队列数×队列的最大长度（字节）。按表15-4中给出的数据，Linux默认配置的最大消息数（系统范围内）是262 144。（即使一个消息可能包含0字节数据，Linux也将其处理为如同包含1字节那样，其目的是限制队列中的消息数。）

表15-4 影响消息队列的系统限制

说　明	典 型 值			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
可发送最长消息的字节数	16 384	8 192	notsup	2 048
一个特定队列的最大字节数（亦即队列中所有消息之和）	2 048	16 384	notsup	4 096
系统中最大消息队列数	40	16	notsup	50
系统中最大消息数	40	derived	notsup	40

回忆表15-1，Mac OS X 10.3不支持XSI消息队列。因为Mac OS X部分地基于FreeBSD，而FreeBSD支持消息队列，所以使Mac OS X支持消息队列是有可能的。确实，一个良好的因特网搜索引擎将提供指针，指向Max OS X的XSI消息队列的第三方端口。

调用的第一个函数通常是msgget，其功能是打开一个现存队列或创建一个新队列。 523

```
#include <sys/msg.h>

int msgget(key_t key, int flag);
```

返回值：若成功则返回消息队列ID，若出错则返回-1

15.6.1节说明了将*key*变换成一个标识符的规则，并且讨论是否创建一个新队列或访问一个现存队列。当创建一个新队列时，初始化msqid_ds结构的下列成员:

- ipc_perm结构按15.6.2节中所述进行初始化。该结构中mode成员按*flag*中的相应权限位

设置。这些权限用表15-2中的常量指定。

- msg_qnum、msg_lspid、msg_lrpid、msg_stime和msg_rtime都设置为0。
- msg_ctime设置为当前时间。
- msg_qbytes设置为系统限制值。

若执行成功，msgget返回非负队列ID。此后，该值就可被用于其他三个消息队列函数。

msgctl函数对队列执行多种操作。它和另外两个与信号量和共享存储有关的函数（semctl和shmctl）是XSI IPC的类似于ioctl的函数（亦即垃圾桶函数）。

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

返回值：若成功则返回0，若出错则返回−1

*cmd*参数说明对由*msqid*指定的队列要执行的命令：

IPC_STAT　取此队列的msqid_ds结构，并将它存放在*buf*指向的结构中。

IPC_SET　按由*buf*指向结构中的值，设置与此队列相关结构中的下列四个字段：msg_perm.uid、msg_perm.gid、msg_perm.mode和msg_qbytes。此命令只能由下列两种进程执行：一种是其有效用户ID等于msg_perm.cuid或msg_perm.uid；另一种是具有超级用户特权的进程。只有超级用户才能增加msg_qbytes的值。

IPC_RMID　从系统中删除该消息队列以及仍在该队列中的所有数据。这种删除立即生效。仍在使用这一消息队列的其他进程在它们下一次试图对此队列进行操作时，将出错返回EIDRM。此命令只能由下列两种进程执行：一种是其有效用户ID等于msg_perm.cuid或msg_perm.uid；另一种是具有超级用户特权的进程。

524

这三条命令（IPC_STAT、IPC_SET和IPC_RMID）也可用于信号量和共享存储。

调用msgsnd将数据放到消息队列中。

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

返回值：若成功则返回0，若出错则返回−1

正如前面提及的，每个消息都由三部分组成，它们是：正长整型类型字段、非负长度（*nbytes*）以及实际数据字节（对应于长度）。消息总是放在队列尾端。

*ptr*参数指向一个长整型数，它包含了正的整型消息类型，在其后紧跟着消息数据。（若*nbytes*是0，则无消息数据。）若发送的最长消息是512字节，则可定义下列结构：

```
struct mymesg {
  long  mtype;       /* positive message type */
  char  mtext[512];  /* message data, of length nbytes */
};
```

于是，*ptr*就是一个指向mymesg结构的指针。接收者可以使用消息类型以非先进先出的次序取消息。

某些平台既支持32位环境，又支持64位环境。这影响到长整型和指针的大小。例如，在64位SPARC系统中，Solaris允许32位和64位应用同时存在。如果一个32位应用经由管道或套接字要与64位

> 应用交换此结构，那么，因为在32位应用中，长整型的大小是4字节，而在64位应用中，长整型的大小是8字节，这就造成了问题。这意味着，32位应用期望mtext字段在结构起始地址后的第4个字节处开始，而64位应用则期望mtext字段在结构起始地址后的第8个字节处开始。在这种情况下，64位应用的mtype字段的一部分会被32位应用视为mtext字段的组成部分，而32位应用的mtext字段的头4个字节会被64位应用解释为mtype字段的组成部分。
>
> 但是，对XSI消息队列而言，这种问题是不会出现的。Solaris实现IPC系统调用的32位版本和64位版本，但两者的入口点不同。这些系统调用知道如何处理32位应用与64位应用的通信操作，并对类型字段作特殊处理以避免它干扰消息的数据部分。唯一可能出问题的是，当64位应用向32位应用发送一消息时，如果它在8字节类型字段中设置的值大于32位应用中4字节类型字段可表示的值，那么32位应用在其mtype字段中得到的是一个截短了的值，于是也就丢失了信息。

参数*flag*的值可以指定为IPC_NOWAIT。这类似于文件I/O的非阻塞I/O标志（见14.2节）。若消息队列已满（或者是队列中的消息总数等于系统限制值，或队列中的字节总数等于系统限制值），则指定IPC_NOWAIT使得msgsnd立即出错返回EAGAIN。如果没有指定IPC_NOWAIT，则进程阻塞直到下述情况出现为止：有空间可以容纳要发送的消息；从系统中删除了此队列；或捕捉到一个信号，并从信号处理程序返回。在第二种情况下，返回EIDRM（“标识符被删除”）。最后一种情况则返回EINTR。

525

注意，对删除消息队列的处理不是很完善。因为对每个消息队列并没有设置一个引用计数器（对打开文件则有这种计数器），所以删除一个队列会造成仍在使用这一队列的进程在下次对队列进行操作时出错返回。信号量机制也以同样方式处理其删除。相反，删除一个文件时，要等到使用该文件的最后一个进程关闭了它的文件描述符后，才能删除文件中的内容。

当msgsnd成功返回，与消息队列相关的msqid_ds结构得到更新，以标明发出该调用的进程ID（msg_lspid）、进行该调用的时间（msg_stime），并指示队列中增加了一条消息（msg-qnum）。

msgrcv从队列中取用消息：

```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);

                    返回值：若成功则返回消息的数据部分的长度，若出错则返回-1
```

如同msgsnd中一样，*ptr*参数指向一个长整型数（返回的消息类型存放在其中），跟随其后的是存放实际消息数据的缓冲区。*nbytes*说明数据缓冲区的长度。若返回的消息大于*nbytes*，而且在*flag*中设置了MSG_NOERROR，则该消息被截短。（在这种情况下，不通知我们消息截短了，消息的截去部分被丢弃。）如果没有设置这一标志，而消息又太长，则出错返回E2BIG（消息仍留在队列中）。

参数*type*使我们可以指定想要哪一种消息：

type == 0　返回队列中的第一个消息。

type > 0　返回队列中消息类型为*type*的第一个消息。

type < 0　返回队列中消息类型值小于或等于*type*绝对值的消息，如果这种消息有若干个，则取类型值最小的消息。

*type*值非0用于以非先进先出次序读消息。例如，若应用程序对消息赋优先权，那么*type*就

可以是优先权值。如果一个消息队列由多个客户进程和一个服务器进程使用，那么*type*字段可以用来包含客户进程的进程ID（只要进程ID可以存放在长整型中）。

可以指定*flag*值为IPC_NOWAIT，使操作不阻塞。这使得如果没有所指定类型的消息，则msgrcv返回-1，errno设置为ENOMSG。如果没有指定IPC_NOWAIT，则进程阻塞直至如下情况出现才终止：有了指定类型的消息；从系统中删除了此队列（出错则返回-1且errno置为EIDRM）；或捕捉到一个信号并从信号处理程序返回（msgrcv返回-1，errno设置为EINTR）。

msgrcv成功执行时，内核更新与该消息队列相关联的msqid_ds结构，以指示调用者的进程ID（msg_lrpid）和调用时间（msg_rtime），并将队列中的消息数（msg_qnum）减1。

526

实例：消息队列与流管道的耗时比较

如若需要客户进程和服务器进程之间的双向数据流，可以使用消息队列或全双工管道。（回忆表15-1，通过UNIX域套接字机制（17.3节），全双工管道是可用的，而某些平台通过pipe函数提供全双工管道。）

表15-5显示了在Solaris上三种技术在时间方面的比较，这三种技术是：消息队列、基于STREAMS的管道和UNIX域套接字。测试程序先创建IPC通道，调用fork，然后从父进程向子进程发送约200MB数据。数据发送的方式是：对于消息队列，调用100 000次msgsnd，每个消息长度为2 000字节；对于基于STREAMS的管道，调用100 000次write，每次写2 000字节。时间都以秒为单位。

表15-5　在Solaris上三种IPC的时间比较

操　作	用　户	系　统	时　钟
消息队列	0.57	3.63	4.22
STREAMS管道	0.50	3.21	3.71
UNIX域套接字	0.43	4.45	5.59

从这些数字中可见，消息队列原来的实施目的是提供比一般IPC更高速度的进程通信方法，但现在与其他形式的IPC相比，在速度方面已经没有什么差别了（事实上，基于STREAMS的管道快于消息队列）。（在原来实施消息队列时，唯一的其他形式IPC是半双工管道。）考虑到使用消息队列具有的问题（见15.6.4节），我们得出的结论是，在新的应用程序中不应当再使用它们。　□

15.8　信号量

信号量（semaphore）与已经介绍过的IPC机构（管道、FIFO以及消息列队）不同。它是一个计数器，用于多进程对共享数据对象的访问。

> Single UNIX Specification在其实时扩展的信号量选项中，包括了信号量接口的替代集。本书不讨论这种接口。

为了获得共享资源，进程需要执行下列操作：

(1) 测试控制该资源的信号量。

(2) 若此信号量的值为正，则进程可以使用该资源。进程将信号量值减1，表示它使用了一个资源单位。

(3) 若此信号量的值为0，则进程进入休眠状态，直至信号量值大于0。进程被唤醒后，它返回至第(1)步。

527

当进程不再使用由一个信号量控制的共享资源时，该信号量值增1。如果有进程正在休眠等待此信号量，则唤醒它们。

为了正确地实现信号量，信号量值的测试及减1操作应当是原子操作。为此，信号量通常是在内核中实现的。

常用的信号量形式被称为*二元信号量*或*双态信号量*（binary semaphore）。它控制单个资源，初始值为1。但是一般而言，信号量的初值可以是任一正值，该值说明有多少个共享资源单位可供共享应用。

遗憾的是，XSI的信号量与此相比要复杂得多。三种特性造成了这种并非必要的复杂性：

(1) 信号量并非是单个非负值，而必需将信号量定义为含有一个或多个信号量值的集合。当创建一个信号量时，要指定该集合中信号量值的数量。

(2) 创建信号量（semget）与对其赋初值（semctl）分开。这是一个致命的弱点，因为不能原子地创建一个信号量集合，并且对该集合中的各个信号量值赋初值。

(3) 即使没有进程正在使用各种形式的XSI IPC，它们仍然是存在的。有些程序在终止时并没有释放已经分配给它的信号量，所以我们不得不为这种程序担心。下面将要说明的undo功能就是假定要处理这种情况的。

内核为每个信号量集合设置了一个semid_ds结构：

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  .
  .
  .
};
```

Single UNIX Specification定义了上面所示的各字段，但是具体实现可在semid_ds结构中定义添加的成员。

每个信号量由一个无名结构表示，它至少包含下列成员：

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /* # processes awaiting semval>curval */
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  .
  .
  .
};
```

528

表15-6列出了影响信号量集合的系统限制（见15.6.3节）。

要获得一个信号量ID，要调用的第一个函数是semget。

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

返回值：若成功则返回信号量ID，若出错则返回−1

表15-6　影响信号量的系统限制

说　明	典　型　值			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
任一信号量的最大值	32 767	32 767	32 767	32 767
任一信号量的最大的终止时调整值	16 384	32 767	16 384	16 384
系统中信号量集的最大数	10	128	87 381	10
系统中信号量的最大数	60	32 000	87 381	60
每个信号量集中的最大信号量数	60	250	87 381	25
系统中undo结构的最大数	30	32 000	87 381	30
每个undo结构中的最大undo项数	10	32	10	10
每个semop调用中的最大操作项数	100	32	100	10

15.6.1节说明了将*key*变换为标识符的规则，讨论了是否创建一个新集合，或是引用一个现存的集合。创建一个新集合时，对semid_ds结构的下列成员赋初值：

- 按15.6.2节中所述，对ipc_perm结构赋初值。该结构中的mode被设置为*flag*中的相应权限位。这些权限是用表15-2中的常量设置的。
- sem_otime设置为0。
- sem_ctime设置为当前时间。
- sem_nsems设置为*nsems*。

*nsems*是该集合中的信号量数。如果是创建新集合（一般在服务器进程中），则必须指定*nsems*。如果引用一个现存的集合（一个客户进程），则将*nsems*指定为0。

semctl函数包含了多种信号量操作。

```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd,
           ... /* union semun arg */);
```

返回值：（见下）

注意，依赖于所请求的命令，第四个参数是可选的，如果使用该参数，则其类型是semun，它是多个特定命令参数的联合（union）：

529

```
union semun {
  int              val;    /* for SETVAL */
  struct semid_ds  *buf;   /* for IPC_STAT and IPC_SET */
  unsigned short   *array; /* for GETALL and SETALL */
};
```

注意，这是一个联合，而非指向联合的指针。

cmd参数指定下列10种命令中的一种，在*semid*指定的信号量集合上执行此命令。其中有5条命令是针对一个特定的信号量值的，它们用*semnum*指定该信号量集合中的一个成员。*semnum*值在0和*nsems*−1之间（包括0和*nsems*−1）。

IPC_STAT　对此集合取semid_ds结构，并存放在由*arg.buf*指向的结构中。

IPC_SET　按由*arg.buf*指向结构中的值设置与此集合相关结构中的下列三个字段值：sem_perm.uid、sem_perm.gid和sem_perm.mode。此命令只能由下列两种进程执行：一种是其有效用户ID等于sem_perm.cuid或

sem_perm.uid的进程;另一种是具有超级用户特权的进程。

IPC_RMID 从系统中删除该信号量集合。这种删除是立即发生的。仍在使用此信号量集合的其他进程在它们下次试图对此信号量集合进行操作时，将出错返回EIDRM。此命令只能由下列两种进程执行：一种是其有效用户ID等于sem_perm.cuid或sem_perm.uid的进程；另一种是具有超级用户特权的进程。

GETVAL 返回成员*semnum*的semval值。

SETVAL 设置成员*semnum*的semval值。该值由*arg.val*指定。

GETPID 返回成员*semnum*的sempid值。

GETNCNT 返回成员*semnum*的semncnt值。

GETZCNT 返回成员*semnum*的semzcnt值。

GETALL 取该集合中所有信号量的值，并将它们存放在由*arg.array*指向的数组中。

SETALL 按*arg.array*指向的数组中的值，设置该集合中所有信号量的值。

对于除GETALL以外的所有GET命令，semctl函数都返回相应的值。其他命令的返回值为0。

函数semop自动执行信号量集合上的操作数组，这是个原子操作。

```
#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

返回值：若成功则返回0，若出错则返回-1

530

参数*semoparray*是一个指针，它指向一个信号量操作数组，信号量操作由sembuf结构表示：

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

参数*nops*规定该数组中操作的数量（元素数）。

对集合中每个成员的操作由相应的sem_op值规定。此值可以是负值、0或正值。（下面的讨论将提到信号量的undo标志。此标志对应于相应sem_flg成员的SEM_UNDO位。）

(1) 最易于处理的情况是sem_op为正。这对应于进程释放占用的资源数。sem_op值加到信号量的值上。如果指定了undo标志，则也从该进程的此信号量调整值中减去sem_op。

(2) 若sem_op为负，则表示要获取由该信号量控制的资源。

如若该信号量的值大于或等于sem_op的绝对值（具有所需的资源），则从信号量值中减去sem_op的绝对值。这保证信号量的结果值大于或等于0。如果指定了undo标志，则sem_op的绝对值也加到该进程的此信号量调整值上。

如果信号量值小于sem_op的绝对值（资源不能满足要求），则：

(a) 若指定了IPC_NOWAIT，则semop出错返回EAGAIN。

(b) 若未指定IPC_NOWAIT，则该信号量的semncnt值加1（因为调用进程将进入休眠状态），然后调用进程被挂起直至下列事件之一发生：

(i) 此信号量变成大于或等于sem_op的绝对值（即某个进程已释放了某些资源）。此信号量的semncnt值减1（因为已结束等待），并且从信号量值中减去sem_op的绝对值。如果指定了undo标志，则sem_op的绝对值也加到该进程的此信号量调整值上。

(ii) 从系统中删除了此信号量。在此情况下，函数出错则返回EIDRM。

531

(iii) 进程捕捉到一个信号，并从信号处理程序返回。在此情况下，此信号量的semncnt值减1（因为调用进程不再等待），并且函数出错返回EINTR。

(3) 若sem_op为0，这表示调用进程希望等待到该信号量值变成0。

如果信号量值当前是0，则此函数立即返回。

如果信号量值非0，则：

(a) 若指定了IPC_NOWAIT，则出错返回EAGAIN。

(b) 若未指定IPC_NOWAIT，则该信号量的semzcnt值加1（因为调用进程将进入休眠状态），然后调用进程被挂起，直至下列事件之一发生为止：

(i) 此信号量值变成0。此信号量的semzcnt值减1（因为调用进程已结束等待）。

(ii) 从系统中删除了此信号量。在此情况下，函数出错返回EIDRM。

(iii) 进程捕捉到一个信号，并从信号处理程序返回。在此情况下此信号量的semzcnt值减1（因为调用进程不再等待），并且函数出错返回EINTR。

semop函数具有原子性，它或者执行数组中的所有操作，或者什么也不做。

exit时的信号量调整

正如前面提到的，如果在进程终止时，它占用了经由信号量分配的资源，那么就会成为一个问题。无论何时，只要为信号量操作指定了SEM_UNDO标志，然后分配资源（sem_op值小于0），那么内核就会记住对于该特定信号量，分配给调用进程多少资源（sem_op的绝对值）。当该进程终止时，不论自愿或者不自愿，内核都将检验该进程是否还有尚未处理的信号量调整值，如果有，则按调整值对相应量值进行处理。

如果用带SETVAL或SETALL命令的semctl设置一信号量的值，则在所有进程中，对于该信号量的调整值都设置为0。

实例：信号量与记录锁的耗时比较

如果多个进程共享一个资源，则可使用信号量或记录锁。对这两种技术在时间上的差别进行比较是有益的。

532

若使用信号量，则先创建一个包含一个成员的信号量集合，然后对该信号量值赋初值1。为了分配资源，以sem_op为-1调用semop；为了释放资源，则以sem_op为+1调用semop。对每个操作都指定SEM_UNDO，以处理在未释放资源条件下进程终止的情况。

若使用记录锁，则先创建一个空文件，并且用该文件的第一个字节（无需存在）作为锁字节。为了分配资源，先对该字节获得一个写锁；释放该资源时，则对该字节解锁。记录锁的性质确保了当一个锁的属主进程终止时，内核会自动释放该锁。

表15-7显示了在Linux上使用这两种不同技术进行锁操作所需的时间。在每一种情况中，资源都被分配，然后释放，如此循环10 000次。这同时由三个不同的进程执行。表15-7中所示的时间是三个进程的总计，单位是秒。

表15-7 信号量锁和记录锁的时间比较

操　作	用　户	系　统	时　钟
带undo的信号量	0.38	0.48	0.86
建议性记录锁	0.41	0.95	1.36

在Linux上，记录锁与信号量锁相比，在时间上要多耗用约60%。

虽然记录锁慢于信号量锁，但如果只需锁一个资源（例如共享存储段）并且不需要使用XSI信号量的所有花哨的功能，则宁可使用记录锁。理由是使用简易，且进程终止时系统会处理任何遗留下来的锁。□

15.9 共享存储

共享存储允许两个或更多进程共享一给定的存储区。因为数据不需要在客户进程和服务器进程之间复制，所以这是最快的一种IPC。使用共享存储时要掌握的唯一窍门是多个进程之间对一给定存储区的同步访问。若服务器进程正在将数据放入共享存储区，则在它做完这一操作之前，客户进程不应当去取这些数据。通常，信号量被用来实现对共享存储访问的同步。（不过正如前节最后部分所述，记录锁也可用于这种场合。）

> Single UNIX Specification在其实时扩展的共享存储对象选项中，包括了访问共享存储的一套替代接口。在本书中不涉及该实时扩展。

内核为每个共享存储段设置了一个shmid_ds结构。 533

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
    .
    .
    .
};
```

（按照支持共享存储段的需要，每种实现会在shmid_ds结构中增加其他成员。）

shmatt_t类型定义为不带符号整型，它至少与unsigned short一样大。表15-8列出了影响共享存储的系统限制（见15.6.3节）。

表15-8 影响共享存储的系统限制

说明	典型值			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
共享存储段的最大字节数	33 554 432	33 554 432	4 194 304	8 388 608
共享存储段的最小字节数	1	1	1	1
系统中共享存储段的最大段数	192	4 096	32	100
每个进程共享存储段的最大段数	128	4 096	8	6

为获得一个共享存储标识符，调用的第一个函数通常是shmget。

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

返回值：若成功则返回共享存储ID，若出错则返回−1

15.6.1节说明了将*key*变换成一个标识符的规则，以及是创建一个新共享存储段还是引用一个现存的共享存储段。当创建一个新段时，初始化shmid_ds结构的下列成员：

- ipc_perm结构按15.6.2节中所述进行初始化。该结构中的mode成员按*flag*中的相应权限位设置。这些权限用表15-2中的常量指定。
- shm_lpid、shm_nattach、shm_atime、以及shm_dtime都设置为0。
- shm_ctime设置为当前时间。
- shm_segsz设置为请求的长度（*size*）。

534

参数*size*是该共享存储段的长度（单位：字节）。实现通常将其向上取为系统页长的整数倍。但是，若应用指定的*size*值并非系统页长的整数倍，那么最后一页的余下部分是不可使用的。如果正在创建一个新段（一般是在服务器进程中），则必须指定其*size*。如果正在引用一个现存的段（一个客户进程），则将*size*指定为0。当创建一新段时，段内的内容初始化为0。

shmctl函数对共享存储段执行多种操作。

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

                                返回值：若成功则返回0，若出错则返回-1
```

*cmd*参数指定下列5种命令中一种，使其在*shmid*指定的段上执行。

IPC_STAT　取此段的shmid_ds结构，并将它存放在由*buf*指向的结构中。

IPC_SET　按*buf*指向结构中的值设置与此段相关结构中的下列三个字段：shm_perm.uid、shm_perm.gid以及shm_perm.mode。此命令只能由下列两种进程执行：一种是其有效用户ID等于shm_perm.cuid或shm_perm.uid的进程；另一种是具有超级用户特权的进程。

IPC_RMID　从系统中删除该共享存储段。因为每个共享存储段有一个连接计数（shmid_ds结构中的shm_nattch字段），所以除非使用该段的最后一个进程终止或与该段脱节，否则不会实际上删除该存储段。不管此段是否仍在使用，该段标识符立即被删除,所以不能再用shmat与该段连接。此命令只能由下列两种进程执行:一种是其有效用户ID等于shm_perm.cuid或shm_perm.uid的进程；另一种是具有超级用户特权的进程。

Linux和Solaris提供了下列另外两种命令，但它们并非Single UNIX Specification的组成部分：

SHM_LOCK　将共享存储段锁定在内存中。此命令只能由超级用户执行。

SHM_UNLOCK　解锁共享存储段。此命令只能由超级用户执行。

535

一旦创建了一个共享存储段，进程就可调用shmat将其连接到它的地址空间中。

```
#include <sys/shm.h>

void *shmat(int shmid, const void *addr, int flag);

                返回值：若成功则返回指向共享存储的指针，若出错则返回-1
```

共享存储段连接到调用进程的哪个地址上与*addr*参数以及在*flag*中是否指定SHM_RND位有关。

- 如果*addr*为0，则此段连接到由内核选择的第一个可用地址上。这是推荐的使用方式。
- 如果*addr*非0，并且没有指定SHM_RND，则此段连接到*addr*所指定的地址上。

- 如果*addr*非0，并且指定了SHM_RND，则此段连接到（*addr*−(*addr* mod ulus SHMLBA)）所表示的地址上。SHM_RND命令的意思是“取整”。SHMLBA的意思是“低边界地址倍数”，它总是2的乘方。该算式是将地址向下取最近1个SHMLBA的倍数。

除非只计划在一种硬件上运行应用程序（这在当今是不大可能的），否则不应指定共享段所连接到的地址。所以一般应指定*addr*为0，以便由内核选择地址。

如果在*flag*中指定了SHM_RDONLY位，则以只读方式连接此段。否则以读写方式连接此段。

shmat的返回值是该段所连接的实际地址，如果出错则返回−1。如果shmat成功执行，那么内核将使该共享存储段shmid_ds结构中的shm_nattch计数器值加1。

当对共享存储段的操作已经结束时，则调用shmdt脱接该段。注意，这并不从系统中删除其标识符以及其数据结构。该标识符仍然存在，直至某个进程（一般是服务器进程）调用shmctl（带命令IPC_RMID）特地删除它。

```
#include <sys/shm.h>

int shmdt(void *addr);
```

返回值：若成功则返回0，若出错则返回−1

*addr*参数是以前调用shmat时的返回值。如果成功，shmdt将使相关shmid_ds结构中的shm_nattch计数器值减1。

内核将以地址0连接的共享存储段放在什么位置上与系统密切相关。程序清单15-11打印一些信息，它们与特定系统将各种不同类型的数据放在什么位置有关。

536

程序清单15-11 打印各种不同类型的数据所存放的位置

```
#include "apue.h"
#include <sys/shm.h>

#define ARRAY_SIZE  40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    0600    /* user read/write */

char    array[ARRAY_SIZE];  /* uninitialized data = bss */

int
main(void)
{
    int     shmid;
    char    *ptr, *shmptr;

    printf("array[] from %lx to %lx\n", (unsigned long)&array[0],
      (unsigned long)&array[ARRAY_SIZE]);
    printf("stack around %lx\n", (unsigned long)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %lx to %lx\n", (unsigned long)ptr,
      (unsigned long)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
```

```
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
    printf("shared memory attached from %lx to %lx\n",
      (unsigned long)shmptr, (unsigned long)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

在一个基于Intel的Linux系统上运行此程序，其输出如下：

```
$ ./a.out
array[] from 804a080 to 8053cc0
stack around bffff9e4
malloced from 8053cc8 to 806c368
shared memory attached from 40162000 to 4017a6a0
```

图15-13描绘了这种情况，这与图7-3中所示的典型存储区布局类似。注意，共享存储段紧靠在栈之下。 □

537

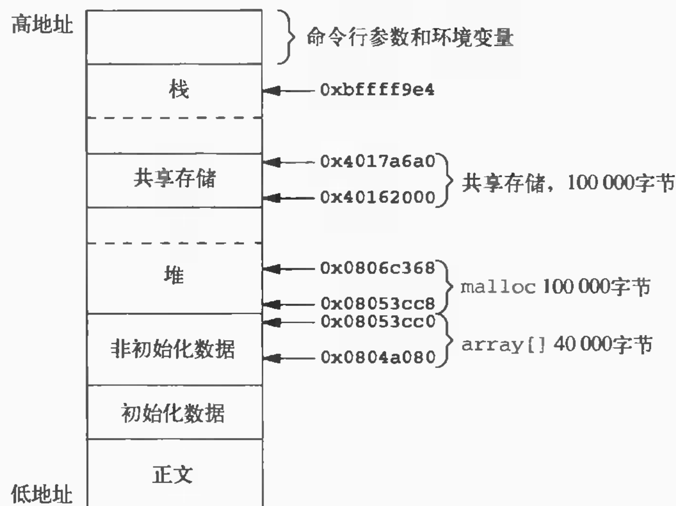


图15-13 在基于Intel的Linux系统上的存储区布局

14.9节中曾说明mmap函数可将一个文件的若干部分映射至进程地址空间。这在概念上类似于用shmat XSI IPC函数连接一共享存储段。两者之间的主要区别是，用mmap映射的存储段是与文件相关联的，而XSI共享存储段则并无这种关联。

实例：/dev/zero的存储映射

共享存储可由不相关的进程使用。但如果进程是相关的，则某些实现提供了一种不同的技术。

> 下面说明的技术用于FreeBSD 5.2.1、Linux 2.4.22和Solaris 9。Mac OS X 10.3当前并不支持将字符设备映射至进程地址空间。

在读设备/dev/zero时，该设备是0字节的无限资源。它也接收写向它的任何数据，但又忽略这些数据。我们对此设备作为IPC的兴趣在于，当对其进行存储映射时，它具有一些特殊

的性质：

- 创建一个未名存储区，其长度是mmap的第二个参数，将其向上取整为系统的最近页长。
- 存储区都初始化为0。
- 如果多个进程的共同祖先进程对mmap指定了MAP_SHARED标志，则这些进程可共享此存储区。

程序清单15-12是使用此特殊设备的一个例子。

程序清单15-12 在父、子进程间使用/dev/zero存储映射I/O的IPC

538

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE        sizeof(long)    /* size of shared memory area */

static int
update(long *ptr)
{
    return((*ptr)++);   /* return value before increment */
}

int
main(void)
{
    int     fd, i, counter;
    pid_t   pid;
    void    *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {           /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {                        /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("child: expected %d, got %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

539

它打开此/dev/zero设备，然后指定长整型的长度调用mmap。注意，一旦存储区映射成功，就关闭此设备。然后，进程创建一个子进程。因为在调用mmap时指定了MAP_SHARED，所以一个进程写到存储映射区的数据可由另一进程见到。（如果已指定MAP_PRIVATE，则此示例程序不能工作。）

然后，父、子进程交替运行，使用8.9节中的同步函数各自对共享存储映射区中的长整型数加1。存储映射区由mmap初始化为0。父进程先对它进行增1操作，使其成为1，然后子进程对其进行增1操作，使其成为2，然后父进程使其成为3……注意，当在update函数中对长整型值增1时，因为增加的是其值，而不是指针，所以必须使用括号。

以上述方式使用/dev/zero的优点是：在调用mmap创建映射区之前，无需存在一个实际文件。映射/dev/zero自动创建一个指定长度的映射区。这种技术的缺点是：它只在相关进程间起作用。但在相关进程之间使用线程（第11章和第12章）可能更为简单、有效。注意，无论使用哪一种技术，都需对共享数据进行同步访问。 □

实例：匿名存储映射

很多实现提供了一种类似于/dev/zero的设施，称为匿名存储映射。为了使用这种功能，在调用mmap时指定MAP_ANON标志，并将文件描述符指定为-1。结果得到的区域是匿名的（因为它并不通过一个文件描述符与一个路径名相结合），并且创建一个可与后代进程共享的存储区。

> 本书讨论的四种平台都支持匿名存储映射。但是注意，Linux为此定义了MAP_ANONYMOUS标志，并将MAP_ANON标志定义为与它相同的值以改善应用的可移植性。

为使程序清单15-12所示程序应用这种特征，对它做了三处修改：一是删除了对于/dev/zero的open语句，二是删除了对于fd的close语句，三是将mmap调用修改成

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                  MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

的形式。在此调用中，指定了MAP_ANON标志，并将文件描述符取为-1。程序的其余部分则没有改变。 □

最后两个例子说明了在多个相关进程之间如何使用共享存储段。如果在无关进程之间使用共享存储段，那么有两种替换的方法。其一是应用程序使用XSI共享存储函数；另一种是使用mmap将同一文件映射至它们的地址空间，为此使用MAP_SHARED标志。 540

15.10 客户进程-服务器进程属性

下面详细说明客户进程和服务器进程的某些属性，这些属性受到它们之间所使用的IPC类型的影响。最简单的关系类型是使客户调用fork然后调用exec执行所希望的服务器进程。在fork之前先创建两个半双工管道使数据可在两个方向传输。图15-8是这种形式的一个例子。被执行的服务器程序可能是设置用户ID的程序，这使它具有了特权。服务器进程查看客户进程的实际用户ID就可以决定客户进程的身份。（回忆8.10节，从中可了解到在exec前后实际用户ID和实际组ID并没有改变。）

在这种安排下，可以构筑一个*开放式服务器*（open server）。（17.5节提供了这种客户和服务器的一种实现）。它为客户进程打开文件而不是客户进程自己调用open函数。这样就可以在

正常的UNIX用户/组/其他权限之上或之外，增加附加的权限检查。假定服务器进程执行的是设置用户ID程序，这给予了它附加的权限（很可能是root权限）。服务器进程用客户进程的实际用户ID以决定是否给予它对所请求文件的访问权限。使用这种方式，可以构筑一个服务器进程，它允许某种用户获得通常没有的访问权限。

在此例子中，因为服务器进程是父进程的子进程，所以它能做的一切是将文件内容传送给父进程。这种方式对普通文件完全够用，但是对特殊设备文件却不能工作。我们希望能做的是使服务器进程打开所要的文件，并送回文件描述符。但是实际情况却是父进程可向子进程传送打开文件描述符，而子进程则不能向父进程传回文件描述符（除非使用将在第17章介绍的专门编程技术）。

图15-12中示出了另一种类型的服务器进程。这种服务器进程是一个守护进程，所有客户进程用某种形式的IPC与其联系。对于这种形式的客户进程-服务器进程关系，不能使用管道。要求使用命名的IPC，例如FIFO或消息队列。对于FIFO，如果服务器进程必须将数据送回客户进程，则对每个客户进程都要有单独使用的FIFO。如果客户进程-服务器进程应用程序只有客户进程向服务器进程发送数据，则只需要一个众所周知的FIFO。（系统V行式打印机假脱机程序使用这种形式的客户进程-服务器进程。客户进程是`lp`(1)命令，服务器进程是`lpsched`守护进程。因为只有从客户进程到服务器进程的数据流，没有任何数据需送回客户进程，所有只需使用一个FIFO。）

使用消息队列则存在多种可能性：

(1) 在服务器进程和所有客户进程之间只使用一个队列，使用消息的类型字段指明谁是消息的接收者。例如，客户进程可以用类型字段1发送它们的消息。在请求之中应包括客户进程的进程ID。此后，服务器进程在发送响应消息时，将类型字段设置为客户进程的进程ID。服务器进程只接收类型字段为1的消息（`msgrcv`的第四个参数），客户进程则只接收类型字段等于它进程ID的消息。

541

(2) 另一种方法是每个客户进程使用一个单独的消息队列。在向服务器进程发送第一个请求之前，每个客户进程先创建它自己的消息队列，创建时使用键`IPC_PRIVATE`。服务器进程也有它自己的队列，其键或标识符是所有客户进程都知道的。客户进程将其第一个请求送到服务器进程的众所周知的队列上，该请求中应包含其客户进程消息队列的队列ID。服务器进程将其第一个响应送至客户进程队列，此后的所有请求和响应都在此队列上交换。

使用这种技术的一个问题是：每个客户进程专用队列通常只有一个消息在其中——或者是对服务器进程的一个请求，或者是对客户进程的响应。这似乎是对有限的系统资源（消息队列）的浪费，为此可以用一个FIFO来代替。另一个问题是服务器进程需从多个队列读消息。对于消息队列，`select`和`poll`都不起作用。

使用消息队列的这两种技术都可以用共享存储段和同步方法（信号量或记录锁）实现。

这种类型的客户进程-服务器进程关系（客户进程和服务器进程是无关系进程）的问题是：服务器进程如何准确地标识客户进程？除非服务器进程正在执行一种非特权操作，否则服务器进程知道客户进程的身份是很重要的。例如，若服务器进程是一个设置用户ID程序，就有这种要求。虽然，所有这几种形式的IPC都经由内核，但是它们并未提供任何措施使内核能够标识发送者。

对于消息队列，如果在客户进程和服务器进程之间使用一个专用队列（于是一次只有一个消息在该队列上），那么队列的`msg_lspid`包含了对方进程的进程ID。但是当客户进程将请求

发送给服务器进程时，我们想要的是客户进程的有效用户ID，而不是它的进程ID。现在还没有一种可移植的方法，在已知进程ID的情况下可以得到有效用户ID。(内核在进程表项中自然地保持有这两种值，但是除非彻底检查内核存储空间，否则已知一个，无法得到另一个。)

我们将在17.3节中使用下列技术,使服务器进程可以标识客户进程。这一技术既可使用FIFO、消息队列或信号量，也可使用共享存储。在下面的说明中假定按图15-12使用了FIFO。客户进程必须创建它自己的FIFO，并且设置该FIFO的文件访问权限，使得只允许用户读，用户写。假定服务器进程具有超级用户特权（或者它很可能并不关心客户进程的真实标识），所以服务器进程仍可读、写此FIFO。当服务器进程在众所周知的FIFO上接收到客户进程的第一个请求时（它应当包含客户进程专用FIFO的标识），服务器进程调用针对客户进程专用FIFO的stat或fstat。服务器进程假设客户进程的有效用户ID是FIFO的所有者（stat结构的st_uid字段）。服务器进程验证该FIFO只有用户读、用户写权限。服务器进程还应检查该FIFO的三个时间量(stat结构中的st_atime，st_mtime和st_ctime字段），要检查它们与当前时间是否很接近（例如不早于当前时间15s或30s）。如果一个有预谋的客户进程可以创建一个FIFO，使另一个用户成为其所有者，并且设置该文件的权限为用户读和用户写，那么在系统中就存在了其他基础性的安全问题。

542

为了用XSI IPC实现这种技术，回想一下与每个消息队列、信号量以及共享存储段相关的ipc_perm结构，其中cuid和cgid字段标识IPC结构的创建者。以FIFO为例，服务器进程应当要求客户进程创建该IPC结构，并使客户进程将访问权限设置为只允许用户读和用户写。服务器进程也应检验与该IPC相关的时间值与当前时间是否很接近（因为这些IPC结构在显式地删除之前一直存在)。

在17.2.2节中，将会看到进行这种身份验证的一种更好的方法，其关键是内核提供客户进程的有效用户ID和有效组ID。STREAMS子系统在进程之间传送文件描述符时可以做到这一点。

15.11　小结

本章详细说明了进程间通信的多种形式：管道、命名管道（FIFO）以及另外三种IPC形式(通常称为XSI IPC)，即消息队列、信号量和共享存储。信号量实际上是同步原语而不是IPC，常用于共享资源（例如共享存储段）的同步访问。对于管道，我们说明了popen函数的实现，说明了协同进程，以及使用标准I/O库缓冲机制时可能遇到的问题。

将消息队列对全双工管道、信号量对记录锁等不同方法的耗时做了比较，然后提出了下列建议：要学会使用管道和FIFO，因为在大量应用程序中仍可有效地使用这两种基本技术。在新的应用程序中，要尽可能避免使用消息队列以及信号量，而应当考虑全双工管道和记录锁，它们使用起来会简单得多。共享存储段有其应用场合，而mmap函数（见14.9节）也能提供同样的功能。

下一章将介绍网络IPC，它们使进程能够跨越计算机的边界进行通信。

习题

15.1　在程序清单15-2父进程代码的末尾，如果删除waitpid前的close，结果将如何？

15.2　在程序清单15-2父进程代码的末尾，如果删除waitpid，结果将如何？

15.3　如果popen函数的参数是一个不存在的命令，这会造成什么结果？编写一段小程序对此进行测试。

15.4 删除程序清单15-9中的信号量处理程序，执行该程序并终止子进程。输入一行后，怎样才能说明父进程是由SIGPIPE终止的？

15.5 将程序清单15-9中进行管道读、写的read和write用标准I/O库代替。

543

15.6 POSIX.1加入waitpid函数的理由之一是，POSIX.1之前的大多数系统不能处理下面的代码。

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

若在这段代码中不使用waitpid函数会如何？用wait代替呢？

15.7 当一个管道被写进程关闭后，解释select和poll如何处理该管道的输入描述符。编两个测试程序，一个用select，另一个用poll，并判断答案是否正确。当一个管道的读端被关闭时，请重做此习题以查看该管道的输出描述符。

15.8 如果popen以*type*为"r"执行*cmdstring*，并将结果写到标准出错输出，结果如何？

15.9 popen函数能使shell执行它的*cmdstring*参数，当cmdstring终止时会产生什么结果？(提示：画出与此相关的所有进程。)

15.10 大多数UNIX系统允许读写FIFO，但是POSIX.1特别声明没有定义为读写而打开FIFO。请用非阻塞方法实现为读写而打开FIFO。

15.11 除非文件包含敏感或机密数据，否则允许其他用户读文件不会造成损害。(不过，窥探别人的文件总归是不良行为。)(但是，如果一个恶意进程读取了被一个服务器进程和几个客户进程使用的消息队列中的一条消息后，会产生什么后果？恶意进程需要知道哪些信息就可以读消息队列？

15.12 编写一段程序完成下面的工作：执行一个循环5次，在每次循环中，创建一个消息队列，打印该队列的标识符，然后删除队列。接着再循环5次，在每次循环中利用键IPC_PRIVATE创建消息队列并将一条消息放在队列中。程序终止后用ipcs(1)查看消息队列。解释队列标识符的变化。

15.13 描述如何在共享存储段中建立一个数据对象的链接列表。列表指针如何保存？

15.14 画出程序清单15-12所示程序运行时下列值随时间变化的曲线图（假定在调用fork后子进程首先运行），这些值是：

(1) 在父进程和子进程中的变量i；

(2) 在共享存储区中长整型的值；

(3) update函数的返回值。

544

15.15 使用15.9节中的XSI共享存储函数代替共享存储映射区，改写程序清单15-12。

15.16 使用15.8节中XSI信号量函数改写程序清单15-12，实现父进程与子进程间的交替。

15.17 使用建议性记录锁改写程序清单15-12，实现父进程与子进程间的交替。

第16章

网络IPC：套接字

16.1 引言

上一章考查了各种UNIX系统所提供的经典进程间通信（IPC）机制：管道、先进先出、消息队列、信号量以及共享内存。通过这些机制，同一台计算机上运行的进程可以相互通信。本章将考查不同计算机（通过网络相连）上运行的进程相互通信的机制：网络进程间通信（network IPC）。

在本章中，将描述套接字网络IPC接口，进程能够使用该接口和其他进程通信。通过该接口，其他进程运行位置是透明的，它们可以在同一台计算机上也可以在不同的计算机上。实际上，这正是套接字接口的目标之一：同样的接口既可以用于计算机间通信又可以用于计算机内通信。尽管套接字接口可以采用许多不同的网络协议，但本章的讨论仅限于因特网事实上的通信标准：TCP/IP协议栈。

POSIX.1所规定的套接字API是基于4.4BSD套接字接口的。尽管这些年有些微小变化，但是当前的套接字接口与20世纪80年代早期4.2BSD中最初引入的接口仍然非常类似。

本章只是对套接字API的概述。Stevens、Fenner和Rudoff[2004]在有关UNIX系统网络编程的权威性文献中详细讨论了套接字接口。

16.2 套接字描述符

套接字是通信端点的抽象。与应用程序要使用文件描述符访问文件一样，访问套接字也需要用套接字描述符。套接字描述符在UNIX系统是用文件描述符实现的。事实上，许多处理文件描述符的函数（如read和write）都可以处理套接字描述符。

要创建一个套接字，可以调用socket函数。

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

返回值：若成功则返回文件（套接字）描述符，若出错则返回-1

参数*domain*（域）确定通信的特性，包括地址格式（在下一小节详细讲述）。表16-1总结了由POSIX.1指定的各个域。各个域有自己的格式表示地址，而表示各个域的常数都以AF_开头，意指地址族（*address family*）。

UNIX域将在17.3节讨论。多数系统还会定义AF_LOCAL域，这是AF_UNIX的别名。AF_UNSPEC域可以代表任何域。历史上，有些平台支持其他网络协议（如AF_IPX为NetWare协议族），但这些协议的域常数没有在POSIX.1 标准中定义。

表16-1　套接字通信域

域	描　述
AF_INET	IPv4因特网域
AF_INET6	IPv6因特网域
AF_UNIX	UNIX域
AF_UNSPEC	未指定

参数*type*确定套接字的类型，进一步确定通信特征。表16-2总结了由POSIX.1 定义的套接字类型，但在实现中可以自由增加对其他类型的支持。

表16-2　套接字类型

类　型	描　述
SOCK_DGRAM	长度固定的、无连接的不可靠报文传递
SOCK_RAW	IP协议的数据报接口（POSIX.1中为可选）
SOCK_SEQPACKET	长度固定、有序、可靠的面向连接报文传递
SOCK_STREAM	有序、可靠、双向的面向连接字节流

546

参数*protocol*通常是零，表示按给定的域和套接字类型选择默认协议。当对同一域和套接字类型支持多个协议时，可以使用*protocol*参数选择一个特定协议。在AF_INET通信域中套接字类型SOCK_STREAM的默认协议是TCP（传输控制协议）。在AF_INET通信域中套接字类型SOCK_DGRAM的默认协议是UDP（用户数据报协议）。

对于数据报（SOCK_DGRAM）接口，与对方通信时是不需要逻辑连接的。只需要送出一个报文，其地址是一个对方进程所使用的套接字。

因此数据报提供了一个无连接的服务。另一方面，字节流（SOCK_STREAM）要求在交换数据之前，在本地套接字和与之通信的远程套接字之间建立一个逻辑连接。

数据报是一种自包含报文。发送数据报近似于给某人邮寄信件。可以邮寄很多信，但不能保证投递的次序，并且可能有些信件丢失在路上。每封信件包含接收者的地址，使这封信件独立于所有其他信件。每封信件可能送达不同的接收者。

相比之下，使用面向连接的协议通信就像与对方打电话。首先，需要通过电话建立一个连接，连接建立好之后，彼此能双向地通信。每个连接是端到端的通信信道。会话中不包含地址信息，就像呼叫的两端存在一个点对点虚拟连接，并且连接本身暗含特定的源和目的地。

对于SOCK_STREAM套接字，应用程序意识不到报文界限，因为套接字提供的是字节流服务。这意味着当从套接字读出数据时，它也许不会返回所有由发送进程所写的字节数。最终可以获得发送过来的所有数据，但也许要通过若干次函数调用得到。

SOCK_SEQPACKET套接字和SOCK_STREAM套接字很类似，但从该套接字得到的是基于报文的服务而不是字节流服务。这意味着从SOCK_SEQPACKET套接字接收的数据量与对方所发送的一致。流控制传输协议（Stream Control Transmission Protocol，SCTP）提供了因特网域上的顺序数据包服务。

SOCK_RAW套接字提供一个数据报接口用于直接访问下面的网络层（在因特网域中为IP）。使用这个接口时，应用程序负责构造自己的协议首部，这是因为传输协议（TCP和UDP等）被绕过了。当创建一个原始套接字时需要有超级用户特权，用以防止恶意程序绕过内建安全机制来创建报文。

调用socket与调用open相类似。在两种情况下，均可获得用于输入/输出的文件描述符。

当不再需要该文件描述符时，调用close来关闭对文件或套接字的访问，并且释放该描述符以便重新使用。

虽然套接字描述符本质上是一个文件描述符，但不是所有参数为文件描述符的函数都可以接受套接字描述符。表16-3总结了到目前为止所讨论的大多数使用文件描述符的函数处理套接字描述符时的行为。未规定的和由实现定义的行为通常意味着函数不能处理套接字描述符。例如，lseek不处理套接字，因为套接字不支持文件偏移量的概念。 547

表16-3 使用文件描述符的函数处理套接字时的行为

函 数	处理套接字时的行为
close（3.3节）	释放套接字
dup, dup2（3.12节）	和一般文件描述符一样复制
fchdir（4.22节）	失败，并且将errno设置为ENOTDIR
fchmod（4.9节）	未规定
fchown（4.11节）	由实现定义
fcntl（3.14节）	支持一些命令，例如F_DUPFD, F_GETFD, F_GETFL, F_GETOWN, F_SETFD, F_SETFL, F_SETOWN
fdatasync, fsync（3.13节）	由实现定义
fstat（4.2节）	支持一些stat结构成员，但如何支持由实现定义
ftruncate（4.13节）	未规定
getmsg, getpmsg（14.4节）	如果套接字由STREAMS实现则可支持，例如在Solaris平台上
ioctl（3.15节）	支持部分命令，依赖于底层设备驱动
lseek（3.6节）	由实现定义（通常是失败并且将errno设为ESPIPE）
mmap（14.9节）	未规定
poll（14.5.2小节）	正常工作
putmsg, putpmsg（14.4节）	如果套接字由STREAMS实现则可支持，例如在Solaris平台上
read（3.7节）和readv（14.7节）	与没有任何标志位的recv（16.5节）等价
select（14.5.1小节）	正常工作
write（3.8节）和writev（14.7节）	与没有任何标志位的send（16.5节）等价

套接字通信是双向的。可以采用函数shutdown来禁止套接字上的输入/输出。

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

返回值：若成功则返回0，若出错则返回-1

如果*how*是SHUT_RD（关闭读端），那么无法从套接字读取数据；如果*how*是SHUT_WR（关闭写端），那么无法使用套接字发送数据；使用SHUT_RDWR则将同时无法读取和发送数据。

能够close（关闭）套接字，为何还使用shutdown呢？理由如下：首先，close只有在最后一个活动引用被关闭时才释放网络端点。这意味着如果复制一个套接字（例如采用dup），套接字直到关闭了最后一个引用它的文件描述符之后才会被释放。而shutdown允许使一个套接字处于不活动状态，无论引用它的文件描述符数目多少。其次，有时只关闭套接字双向传输中的一个方向会很方便。例如，如果想让所通信的进程能够确定数据发送何时结束，可以关闭该套接字的写端，然而通过该套接字读端仍可以继续接收数据。 548

16.3 寻址

上一节中学习了如何创建和销毁一个套接字。在学习用套接字做一些有意义的事情之前，

需要知道如何确定一个目标通信进程。进程的标识有两个部分：计算机的网络地址可以帮助标识网络上想与之通信的计算机，而服务可以帮助标识计算机上特定的进程。

16.3.1 字节序

运行在同一台计算机上的进程相互通信时，一般不用考虑字节的顺序（字节序），字节序是一个处理器架构特性，用于指示像整数这样的大数据类型的内部字节顺序。图16-1显示一个32位整数内部的字节是如何排序的。

如果处理器架构支持大端（big-endian）字节序，那么最大字节地址对应于数字最低有效字节（LSB）上；小端（little-endian）字节序则相反：数字最低字节对应于最小字节地址。注意，不管字节如何排序，数字最高位总是在左边，最低位总是在右边。因此，如果想给一个32位整数赋值`0x04030201`，不管字节如何排序，数字最高位包含4，数字最低位包含1。如果接着想将一个字符指针（`cp`）强制转换到这个整数的地址，将看到字节序带来的不同。在小端字节序的处理器上，`cp[0]`指向数字最低位因而包含1，`cp[3]`指向数字最高位因而包含4。相比较而言，对于大端字节序的处理器，`cp[0]`指向数字最高位因而包含4，`cp[3]`指向数字最低位因而包含1。表16-4总结了本文所讨论的4种平台的字节序。

549

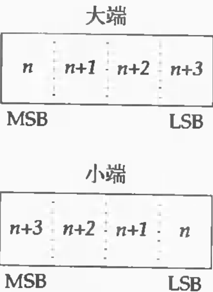


图16-1　32位整数内部的字节序

表16-4　测试平台的字节序

操作系统	处理器架构	字节序
FreeBSD 5.2.1	Intel Pentium	小端
Linux 2.4.22	Intel Pentium	小端
Mac OS X 10.3	PowerPC	大端
Solaris 9	Sun SPARC	大端

> 有些处理器可以配置成大端或小端，使问题变得乱上加乱。

网络协议指定了字节序，因此异构计算机系统能够交换协议信息而不会混淆字节序。TCP/IP协议栈采用大端字节序。应用程序交换格式化数据时，字节序问题就会出现。对于TCP/IP，地址用网络字节序来表示，所以应用程序有时需要在处理器的字节序与网络字节序之间的转换。例如，当打印一个易于阅读的地址时，这种转换是很平常的。

对于TCP/IP应用程序，提供了四个通用函数以实施在处理器字节序和网络字节序之间的转换。

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostint32);
                                        返回值：以网络字节序表示的32位整型数
uint16_t htons(uint16_t hostint16);
                                        返回值：以网络字节序表示的16位整型数
uint32_t ntohl(uint32_t netint32);
                                        返回值：以主机字节序表示的32位整型数
uint16_t ntohs(uint16_t netint16);
                                        返回值：以主机字节序表示的16位整型数
```

h表示“主机（host)”字节序，n表示“网络（network)”字节序。l表示“长（long)”整数（即4个字节），s表示“短（short)”整数（即2个字节）。这4个函数定义在<arpa/inet.h>中，也有比较老的系统将其定义在<netinet/in.h>中。

550

16.3.2 地址格式

地址标识了特定通信域中的套接字端点，地址格式与特定的通信域相关。为使不同格式地址能够被传入到套接字函数，地址被强制转换成通用的地址结构sockaddr表示：

```
struct sockaddr {
  sa_family_t   sa_family;  /* address family */
  char          sa_data[];  /* variable-length address */
   .
   .
   .
};
```

套接字实现可以自由地添加额外的成员并且定义sa_data成员的大小。例如在Linux中，该结构定义如下：

```
struct sockaddr {
  sa_family_t  sa_family;     /* address family */
  char         sa_data[14];   /* variable-length address */
};
```

而在FreeBSD中，该结构定义如下：

```
struct sockaddr {
  unsigned char  sa_len;        /* total length */
  sa_family_t    sa_family;     /* address family */
  char           sa_data[14];   /* variable-length address */
};
```

因特网地址定义在<netinet/in.h>中。在IPv4 因特网域（AF_INET）中，套接字地址用如下结构sockaddr_in表示:

```
struct in_addr {
  in_addr_t       s_addr;         /* IPv4 address */
};

struct sockaddr_in {
  sa_family_t    sin_family;     /* address family */
  in_port_t      sin_port;       /* port number */
  struct in_addr sin_addr;       /* IPv4 address */
};
```

数据类型in_port_t定义成uint16_t。数据类型in_addr_t定义成uint32_t。这些整数类型在<stdint.h>中定义并指定了相应的位数。

与IPv4因特网域（AF_INET）相比较，IPv6因特网域（AF_INET6）套接字地址用如下结构sockaddr_in6表示：

```
struct in6_addr {
  uint8_t         s6_addr[16];    /* IPv6 address */
};

struct sockaddr_in6 {
  sa_family_t     sin6_family;    /* address family */
  in_port_t       sin6_port;      /* port number */
```

551

```
  uint32_t        sin6_flowinfo;  /* traffic class and flow info */
  struct in6_addr sin6_addr;      /* IPv6 address */
  uint32_t        sin6_scope_id;  /* set of interfaces for scope */
};
```

这些是Single UNIX Specification必需的定义，每个实现可以自由地添加额外的字段。例如，在Linux中，sockaddr_in定义如下：

```
struct sockaddr_in {
  sa_family_t     sin_family;     /* address family */
  in_port_t       sin_port;       /* port number */
  struct in_addr  sin_addr;       /* IPv4 address */
  unsigned char   sin_zero[8];    /* filler */
};
```

其中成员sin_zero为填充字段，必须全部被置为0。

注意，尽管sockaddr_in与sockaddr_in6相差比较大，它们均被强制转换成sockaddr结构传入到套接字例程中。在17.3节，将会看到UNIX域套接字地址与上述因特网域套接字地址格式的不同。

有时，需要打印出能被人而不是计算机所理解的地址格式。BSD网络软件中包含了函数inet_addr和inet_ntoa，用于在二进制地址格式与点分十进制字符串表示（a.b.c.d）之间相互转换。这些函数仅用于IPv4地址，但功能相似的两个新函数inet_ntop和inet_pton支持IPv4和IPv6地址。

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr,
                      char *restrict str, socklen_t size);
```

返回值：若成功则返回地址字符串指针，若出错则返回NULL

```
int inet_pton(int domain, const char *restrict str,
              void *restrict addr);
```

返回值：若成功则返回1，若格式无效则返回0，若出错则返回-1

函数inet_ntop将网络字节序的二进制地址转换成文本字符串格式，inet_pton将文本字符串格式转换成网络字节序的二进制地址。参数*domain*仅支持两个值：AF_INET和AF_INET6。

对于inet_ntop，参数*size*指定了用以保存文本字符串的缓冲区（*str*）的大小。两个常数用于简化工作：INET_ADDRSTRLEN定义了足够大的空间来存放表示IPv4地址的文本字符串，INET6_ADDRSTRLEN定义了足够大的空间来存放表示IPv6地址的文本字符串。对于inet_pton，如果*domain*是AF_INET，缓冲区*addr*需要有足够大的空间来存放32位地址，如果*domain*为AF_INET6则需要足够大的空间来存放128位地址。

552

16.3.3 地址查询

理想情况下，应用程序不需要了解套接字地址的内部结构。如果应用程序只是简单地传递类似于sockaddr结构的套接字地址，并且不依赖于任何协议相关的特性，那么可以与提供相同服务的许多不同协议协作。

历史上，BSD网络软件提供接口访问各种网络配置信息。在6.7节，简要地讨论了网络数据文件和用来访问这种信息的函数。在本节，将更加详细地讨论一些细节，并且引入新的函数来

查询寻址信息。

这些函数返回的网络配置信息可能存放在许多地方。它们可以保存在静态文件中（如/etc/hosts，/etc/services等），或者可以由命名服务管理，例如DNS（Domain Name System）或者NIS（Network Information Service）。无论这些信息放在何处，这些函数同样能够访问它们。

通过调用gethostent，可以找到给定计算机的主机信息。

```
#include <netdb.h>

struct hostent *gethostent(void);

                                返回值：若成功则返回指针，若出错则返回NULL

void sethostent(int stayopen);

void endhostent(void);
```

如果主机数据文件没有打开，gethostent会打开它。函数gethostent返回文件的下一个条目。函数sethostent会打开文件，如果文件已经被打开，那么将其回绕。函数endhostent将关闭文件。

当gethostent返回时，得到一个指向hostent结构的指针，该结构可能包含一个静态的数据缓冲区。每次调用gethostent将会覆盖这个缓冲区。数据结构hostent至少包含如下成员：

```
struct hostent {
  char   *h_name;       /* name of host */
  char  **h_aliases;    /* pointer to alternate host name array */
  int     h_addrtype;   /* address type */
  int     h_length;     /* length in bytes of address */
  char  **h_addr_list;  /* pointer to array of network addresses */
  .
  .
  .
};
```

返回的地址采用网络字节序。

两个附加的函数gethostbyname和gethostbyaddr，原来包含在hostent函数里面，现在被认为是过时的，马上将会看到其替代函数。

553

能够采用一套相似的接口来获得网络名字和网络号。

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);

                    以上三个函数的返回值：若成功则返回指针，若出错则返回NULL

void setnetent(int stayopen);

void endnetent(void);
```

结构netent至少包含如下字段：

```
struct netent {
  char     *n_name;      /* network name */
```

```
    char    **n_aliases;   /* alternate network name array pointer */
    int       n_addrtype;  /* address type */
    uint32_t  n_net;       /* network number */
    :
};
```

网络号按照网络字节序返回。地址类型是一个地址族常量（例如AF_INET）。

可以将协议名字和协议号采用以下函数映射。

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);

                    以上所有函数的返回值：若成功则返回指针，若出错则返回NULL

void setprotoent(int stayopen);

void endprotoent(void);
```

POSIX.1定义的结构protoent至少包含如下成员：

```
struct protoent {
  char   *p_name;     /* protocol name */
  char  **p_aliases;  /* pointer to alternate protocol name array */
  int     p_proto;    /* protocol number */
  :
};
```

[554] 服务是由地址的端口号部分表示的。每个服务由一个唯一的、熟知的端口号来提供。采用函数getservbyname可以将一个服务名字映射到一个端口号，函数getservbyport将一个端口号映射到一个服务名，或者采用函数getservent顺序扫描服务数据库。

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);

                    以上所有函数的返回值：若成功则返回指针，若出错则返回NULL

void setservent(int stayopen);

void endservent(void);
```

结构servent至少包含如下成员：

```
struct servent {
  char   *s_name;     /* service name */
  char  **s_aliases;  /* pointer to alternate service name array */
  int     s_port;     /* port number */
  char   *s_proto;    /* name of protocol */
  :
};
```

POSIX.1定义了若干新的函数，允许应用程序将一个主机名字和服务名字映射到一个地址，或者相反。这些函数代替老的函数gethostbyname和gethostbyaddr。

函数getaddrinfo允许将一个主机名字和服务名字映射到一个地址。

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict host,
                const char *restrict service,
                const struct addrinfo *restrict hint,
                struct addrinfo **restrict res);

                        返回值：若成功则返回0，若出错则返回非0错误码

void freeaddrinfo(struct addrinfo *ai);
```

需要提供主机名字、服务名字，或者两者都提供。如果仅仅提供一个名字，另外一个必须是一个空指针。主机名字可以是一个节点名或点分十进制记法表示的主机地址。

函数getaddrinfo返回一个结构addrinfo的链表。可以用freeaddrinfo来释放一个或多个这种结构，这取决于用ai_next字段链接起来的结构有多少。

555

结构addrinfo的定义至少包含如下成员：

```
struct addrinfo {
  int              ai_flags;       /* customize behavior */
  int              ai_family;      /* address family */
  int              ai_socktype;    /* socket type */
  int              ai_protocol;    /* protocol */
  socklen_t        ai_addrlen;     /* length in bytes of address */
  struct sockaddr *ai_addr;        /* address */
  char            *ai_canonname;   /* canonical name of host */
  struct addrinfo *ai_next;        /* next in list */
  .
  .
  .
};
```

根据某些规则，可以提供一个可选的*hint*来选择地址。*hint*是一个用于过滤地址的模板，仅使用ai_family、ai_flags、ai_protocol和ai_socktype字段。剩余的整数字段必须设为零，并且指针字段为空。表16-5总结了在ai_flags中所用的标志，这些标志用来指定如何处理地址和名字。

表16-5 addrinfo结构标志

标 志	描 述
AI_ADDRCONFIG	查询配置的地址类型（IPv4或IPv6）
AI_ALL	查找IPv4和IPv6地址（仅用于AI_V4MAPPED）
AI_CANONNAME	需要一个规范名（而不是别名）
AI_NUMERICHOST	以数字格式返回主机地址
AI_NUMERICSERV	以端口号返回服务
AI_PASSIVE	套接字地址用于监听绑定
AI_V4MAPPED	如果没有找到IPv6地址，则返回映射到IPv6格式的IPv4地址

如果getaddrinfo失败，不能使用perror或strerror来生成错误消息。替代地，调用gai_strerror将返回的错误码转换成错误消息。

```
#include <netdb.h>

const char *gai_strerror(int error);
```

返回值：指向描述错误的字符串的指针

函数getnameinfo将地址转换成主机名或者服务名。

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict addr,
                socklen_t alen, char *restrict host,
                socklen_t hostlen, char *restrict service,
                socklen_t servlen, unsigned int flags);
```

返回值：若成功则返回0，若出错则返回非0值

556

套接字地址（*addr*）被转换成主机名或服务名。如果*host*非空，它指向一个长度为*hostlen*字节的缓冲区用于存储返回的主机名。同样，如果*service*非空，它指向一个长度为*servlen*字节的缓冲区用于存储返回的服务名。

参数*flags*指定一些转换的控制方式，表16-6总结了系统支持的标志。

表16-6　getnameinfo函数标志

标　志	描　述
NI_DGRAM	服务基于数据报而非基于流
NI_NAMEREQD	如果找不到主机名字，将其作为一个错误对待
NI_NOFQDN	对于本地主机，仅返回完全限定域名的节点名字部分
NI_NUMERICHOST	以数字形式而非名字返回主机地址
NI_NUMERICSERV	以数字形式而非名字返回服务地址（即端口号）

程序清单16-1说明了函数getaddrinfo的使用方法。

程序清单16-1　打印主机和服务信息

```
#include "apue.h"
#include <netdb.h>
#include <arpa/inet.h>
#if defined(BSD) || defined(MACOS)
#include <sys/socket.h>
#include <netinet/in.h>
#endif

void
print_family(struct addrinfo *aip)
{
    printf(" family ");
    switch (aip->ai_family) {
    case AF_INET:
        printf("inet");
        break;
    case AF_INET6:
        printf("inet6");
        break;
```

```
    case AF_UNIX:
        printf("unix");
        break;
    case AF_UNSPEC:
        printf("unspecified");
        break;
    default:
        printf("unknown");
    }
}

void
print_type(struct addrinfo *aip)
{
    printf(" type ");
    switch (aip->ai_socktype) {
    case SOCK_STREAM:
        printf("stream");
        break;
    case SOCK_DGRAM:
        printf("datagram");
        break;
    case SOCK_SEQPACKET:
        printf("seqpacket");
        break;
    case SOCK_RAW:
        printf("raw");
        break;
    default:
        printf("unknown (%d)", aip->ai_socktype);
    }
}

void
print_protocol(struct addrinfo *aip)
{
    printf(" protocol ");
    switch (aip->ai_protocol) {
    case 0:
        printf("default");
        break;
    case IPPROTO_TCP:
        printf("TCP");
        break;
    case IPPROTO_UDP:
        printf("UDP");
        break;
    case IPPROTO_RAW:
        printf("raw");
        break;
    default:
        printf("unknown (%d)", aip->ai_protocol);
    }
}

void
print_flags(struct addrinfo *aip)
{
    printf("flags");
    if (aip->ai_flags == 0) {
        printf(" 0");
```

557

558

```
    } else {
        if (aip->ai_flags & AI_PASSIVE)
            printf(" passive");
        if (aip->ai_flags & AI_CANONNAME)
            printf(" canon");
        if (aip->ai_flags & AI_NUMERICHOST)
            printf(" numhost");
#if defined(AI_NUMERICSERV)
        if (aip->ai_flags & AI_NUMERICSERV)
            printf(" numserv");
#endif
#if defined(AI_V4MAPPED)
        if (aip->ai_flags & AI_V4MAPPED)
            printf(" v4mapped");
#endif
#if defined(AI_ALL)
        if (aip->ai_flags & AI_ALL)
            printf(" all");
#endif
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
    struct sockaddr_in  *sinp;
    const char          *addr;
    int                 err;
    char                abuf[INET_ADDRSTRLEN];

    if (argc != 3)
        err_quit("usage: %s nodename service", argv[0]);
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = 0;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], argv[2], &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        print_flags(aip);
        print_family(aip);
        print_type(aip);
        print_protocol(aip);
        printf("\n\thost %s", aip->ai_canonname?aip->ai_canonname:"-");
        if (aip->ai_family == AF_INET) {
            sinp = (struct sockaddr_in *)aip->ai_addr;
            addr = inet_ntop(AF_INET, &sinp->sin_addr, abuf,
                INET_ADDRSTRLEN);
            printf(" address %s", addr?addr:"unknown");
            printf(" port %d", ntohs(sinp->sin_port));
        }
        printf("\n");
    }
    exit(0);
}
```

559

这个程序说明了函数getaddrinfo的使用方法。如果有多个协议为指定的主机提供相应的服务，程序会打印出超过一条的信息。本例中，仅仅打印工作在IPv4（ai_family为AF_INET）上协议的地址信息。如果想将输出限制在AF_INET协议族，可以在hint中设置ai_family字段。

程序在某个测试系统上运行时，得到了如下输出：

```
$ ./a.out harry nfs
flags canon family inet type stream protocol TCP
     host harry address 192.168.1.105 port 2049
flags canon family inet type datagram protocol UDP
     host harry address 192.168.1.105 port 2049
```

□

16.3.4 将套接字与地址绑定

与客户端的套接字关联的地址没有太大意义，可以让系统选一个默认的地址。然而，对于服务器，需要给一个接收客户端请求的套接字绑定一个众所周知的地址。客户端应有一种方法来发现用以连接服务器的地址，最简单的方法就是为服务器保留一个地址并且在/etc/services或者某个名字服务（name service）中注册。

可以用bind函数将地址绑定到一个套接字。

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

返回值：若成功则返回0，若出错则返回−1

对于所能使用的地址有一些限制：

- 在进程所运行的机器上，指定的地址必须有效，不能指定一个其他机器的地址。
- 地址必须和创建套接字时的地址族所支持的格式相匹配。
- 端口号必须不小于1024，除非该进程具有相应的特权（即为超级用户）。
- 一般只有套接字端点能够与地址绑定，尽管有些协议允许多重绑定。

560

对于因特网域，如果指定IP地址为INADDR_ANY，套接字端点可以被绑定到所有的系统网络接口。这意味着可以收到这个系统所安装的所有网卡的数据包。在下一节中将看到，如果调用connect或listen，但没有绑定地址到一个套接字，系统会选一个地址并将其绑定到套接字。

可以调用函数getsockname来发现绑定到一个套接字的地址。

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

返回值：若成功则返回0，若出错则返回−1

调用getsockname之前，设置*alenp*为一个指向整数的指针，该整数指定缓冲区sockaddr的大小。返回时，该整数会被设置成返回地址的大小。如果该地址和提供的缓冲区长度不匹配，则将其截断而不报错。如果当前没有绑定到该套接字的地址，其结果没有定义。

如果套接字已经和对方连接，调用getpeername来找到对方的地址。

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

返回值：若成功则返回0，若出错则返回-1

除了还会返回对方的地址之外，函数getpeername和getsockname一样。

16.4 建立连接

如果处理的是面向连接的网络服务（SOCK_STREAM或SOCK_SEQPACKET），在开始交换数据以前，需要在请求服务的进程套接字（客户端）和提供服务的进程套接字（服务器）之间建立一个连接。可以用connect建立一个连接。

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

返回值：若成功则返回0，若出错则返回-1

561

在connect中所指定的地址是想与之通信的服务器地址。如果*sockfd*没有绑定到一个地址，connect会给调用者绑定一个默认地址。

当连接一个服务器时，出于一些原因，连接可能失败。要连接的机器必须开启并且正在运行，服务器必须绑定到一个想与之连接的地址，并且在服务器的等待连接队列中应有足够的空间（马上将学到这一点）。因此，应用程序必须能够处理connect返回的错误，这些错误可能由一些瞬时变化条件引起。

程序清单16-2显示了一种如何处理瞬时connect错误的方法。这在一个负载很重的服务器上很有可能发生。

程序清单16-2 支持重试的连接

```
#include "apue.h"
#include <sys/socket.h>

#define MAXSLEEP 128

int
connect_retry(int sockfd, const struct sockaddr *addr, socklen_t alen)
{
    int nsec;

    /*
     * Try to connect with exponential backoff.
     */
    for (nsec = 1; nsec <= MAXSLEEP; nsec <<= 1) {
        if (connect(sockfd, addr, alen) == 0) {
            /*
             * Connection accepted.
             */
            return(0);
        }
```

```
            /*
             * Delay before trying again.
             */
            if (nsec <= MAXSLEEP/2)
                sleep(nsec);
        }
        return(-1);
    }
```

这个函数使用了名为**指数补偿**（exponential backoff）的算法。如果调用connect失败，进程就休眠一小段时间然后再尝试，每循环一次增加每次尝试的延迟，直到最大延迟为2分钟。□ 562

如果套接字描述符处于将要在16.8节讨论的非阻塞模式下，那么在连接不能马上建立时，connect将会返回−1，并且将errno设为特殊的错误码EINPROGRESS。应用程序可以使用poll或者select来判断文件描述符何时可写。如果可写，连接完成。

函数connect还可以用于无连接的网络服务（SOCK_DGRAM）。这看起来有点矛盾，实际上却是一个不错的选择。如果在SOCK_DGRAM套接字上调用connect，所有发送报文的目标地址设为connect调用中所指定的地址，这样每次传送报文时就不需要再提供地址。另外，仅能接收来自指定地址的报文。

服务器调用listen来宣告可以接受连接请求。

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);

                                  返回值：若成功则返回0，若出错则返回−1
```

参数*backlog*提供了一个提示，用于表示该进程所要入队的连接请求数量。其实际值由系统决定，但上限由<sys/socket.h>中SOMAXCONN指定。

> Solaris系统忽略<sys/socket.h>中的SOMAXCONN值，具体的上限依赖于每个协议的实现。对于TCP，其默认值为128。

一旦队列满，系统会拒绝多余连接请求，所以*backlog*的值应该基于服务器期望负载和接受连接请求与启动服务的处理能力来选择。

一旦服务器调用了listen，套接字就能接收连接请求。使用函数accept获得连接请求并建立连接。

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);

                    返回值：若成功则返回文件（套接字）描述符，若出错则返回−1
```

函数accept所返回的文件描述符是套接字描述符，该描述符连接到调用connect的客户端。这个新的套接字描述符和原始套接字（*sockfd*）具有相同的套接字类型和地址族。传给accept的原始套接字没有关联到这个连接，而是继续保持可用状态并接受其他连接请求。

如果不关心客户端标识，可以将参数*addr*和*len*设为NULL；否则，在调用accept之前，应将参数*addr*设为足够大的缓冲区来存放地址，并且将*len*设为指向代表这个缓冲区大小的整数的指针。返回时，accept会在缓冲区填充客户端的地址并且更新指针*len*所指向的整数为该地址 563

的大小。

如果没有连接请求等待处理，accept会阻塞直到一个请求到来。如果*sockfd*处于非阻塞模式，accept会返回−1并将errno设置为EAGAIN或EWOULDBLOCK。

> 本文所讨论的所有平台将EAGAIN定义为与EWOULDBLOCK相同。

如果服务器调用accept并且当前没有连接请求，服务器会阻塞直到一个请求到来。另外，服务器可以使用poll或select来等待一个请求的到来。在这种情况下，一个带等待处理的连接请求套接字会以可读的方式出现。

实　例

程序清单16-3显示了一个服务器进程用以分配和初始化套接字的函数。

程序清单16-3　服务器初始化套接字端点

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd;
    int err = 0;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (bind(fd, addr, alen) < 0) {
        err = errno;
        goto errout;
    }
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
        if (listen(fd, qlen) < 0) {
            err = errno;
            goto errout;
        }
    }
    return(fd);

errout:
    close(fd);
    errno = err;
    return(-1);
}
```

564

我们将会看到，TCP关于地址复用有一些奇怪的规则，导致这个例子并不完备。程序清单16-9显示了有关这个函数的另一个版本，该版本可以绕过这些规则，解决眼下版本的主要缺陷。□

16.5　数据传输

既然将套接字端点表示为文件描述符，那么只要建立连接，就可以使用read和write来

通过套接字通信。回忆前面所讲，通过在connect函数里面设置对方地址，数据报套接字也可以“连接”。在套接字描述符上采用read和write是非常有意义的，因为可以传递套接字描述符到那些原先设计为处理本地文件的函数。而且可以安排传递套接字描述符到执行程序的子进程，该子进程并不了解套接字。

尽管可以通过read和write交换数据，但这就是这两个函数所能做的一切。如果想指定选项、从多个客户端接收数据包或者发送带外数据，需要采用六个传递数据的套接字函数中的一个。

三个函数用来发送数据，三个用于接受数据。首先，考查用于发送数据的函数。

最简单的是send，它和write很像，但是可以指定标志来改变处理传输数据的方式。

```
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

返回值：若成功则返回发送的字节数，若出错则返回-1

类似write，使用send时套接字必须已经连接。参数*buf*和*nbytes*与write中的含义一致。

然而，与write不同的是，send支持第四个参数*flags*。两个标志是Single UNIX Specification规定的，但是其他标志通常实现也支持。表16-7总结了这些标志。

表16-7 send套接字调用标志

标志	描述	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_DONTROUTE	勿将数据路由出本地网络		•	•	•	•
MSG_DONTWAIT	允许非阻塞操作（等价于使用O_NONBLOCK）		•	•	•	
MSG_EOR	如果协议支持，此为记录结束	•	•	•	•	
MSG_OOB	如果协议支持，发送带外数据（见16.7节）	•	•	•	•	•

565

如果send成功返回，并不必然表示连接另一端的进程接收数据。所保证的仅是当send成功返回时，数据已经无错误地发送到网络上。

对于支持为报文设限的协议，如果单个报文超过协议所支持的最大尺寸，send失败并将errno设为EMSGSIZE；对于字节流协议，send会阻塞直到整个数据被传输。

函数sendto和send很类似。区别在于sendto允许在无连接的套接字上指定一个目标地址。

```
#include <sys/socket.h>

ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,
               const struct sockaddr *destaddr, socklen_t destlen);
```

返回值：若成功则返回发送的字节数，若出错则返回-1

对于面向连接的套接字，目标地址是忽略的，因为目标地址蕴涵在连接中。对于无连接的套接字，不能使用send，除非在调用connect时预先设定了目标地址，或者采用sendto来提供另外一种发送报文方式。

可以使用不止一个的选择来通过套接字发送数据。可以调用带有msghdr结构的sendmsg来指定多重缓冲区传输数据，这和writev很相像（14.7节）。

```
#include <sys/socket.h>

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

返回值：若成功则返回发送的字节数，若出错则返回-1

POSIX.1定义了msghdr结构，它至少应该有如下成员：

```
struct msghdr {
  void          *msg_name;        /* optional address */
  socklen_t      msg_namelen;     /* address size in bytes */
  struct iovec  *msg_iov;         /* array of I/O buffers */
  int            msg_iovlen;      /* number of elements in array */
  void          *msg_control;     /* ancillary data */
  socklen_t      msg_controllen;  /* number of ancillary bytes */
  int            msg_flags;       /* flags for received message */
  .
  .
  .
};
```

在14.7节可以看到iovec结构。在17.4.2节中可以看到辅助数据的使用。

566

函数recv和read很像，但是允许指定选项来控制如何接收数据。

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

返回值：以字节计数的消息长度，若无可用消息或对方已经按序结束则返回0，若出错则返回-1

表16-8总结了这些标志。Single UNIX Specification只规定了三个标志。

表16-8　recv套接字调用标志

标　志	描　述	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_OOB	如果协议支持，接收带外数据（见16.7节）	•	•	•	•	•
MSG_PEEK	返回报文内容而不真正取走报文	•	•	•	•	•
MSG_TRUNC	即使报文被截断，要求返回的是报文的实际长度			•		
MSG_WAITALL	等待直到所有的数据可用（仅SOCK_STREAM）	•	•	•	•	•

当指定MSG_PEEK标志时，可以查看下一个要读的数据但不会真正取走。当再次调用read或recv函数时会返回刚才查看的数据。

对于SOCK_STREAM套接字，接收的数据可以比请求的少。标志MSG_WAITALL阻止这种行为，除非所需数据全部收到，recv函数才会返回。对于SOCK_DGRAM和SOCK_SEQPACKET套接字，MSG_WAITALL标志没有改变什么行为，因为这些基于报文的套接字类型一次读取就返回整个报文。

如果发送者已经调用shutdown（16.2节）来结束传输，或者网络协议支持默认的顺序关闭并且发送端已经关闭，那么当所有的数据接收完毕后，recv返回0。

如果有兴趣定位发送者，可以使用recvfrom来得到数据发送者的源地址。

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                 struct sockaddr *restrict addr,
                 socklen_t *restrict addrlen);
```

返回值：以字节计数的消息长度，若无可用消息或对方已经按序结束则返回0，若出错则返回−1

567

如果*addr*非空，它将包含数据发送者的套接字端点地址。当调用recvfrom时，需要设置*addrlen*参数指向一个包含*addr*所指的套接字缓冲区字节大小的整数。返回时，该整数设为该地址的实际字节大小。

因为可以获得发送者的地址，recvfrom通常用于无连接套接字。否则，recvfrom等同于recv。

为了将接收到的数据送入多个缓冲区（类似于readv（14.7节）），或者想接收辅助数据（17.4.2节），可以使用recvmsg。

```
#include <sys/socket.h>

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

返回值：以字节计数的消息长度，若无可用消息或对方已经按序结束则返回0，若出错则返回−1

结构msghdr（在sendmsg中见过）被recvmsg用于指定接收数据的输入缓冲区。可以设置参数*flags*来改变recvmsg的默认行为。返回时，msghdr结构中的msg_flags字段被设为所接收数据的各种特征（进入recvmsg时msg_flags被忽略）。从recvmsg中返回的各种可能值总结在表16-9中。可以在第17章中看见使用recvmsg的例子。

表16-9 从recvmsg中返回的msg_flags标志

标 志	描 述	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_CTRUNC	控制数据被截断	•	•	•	•	•
MSG_DONTWAIT	recvmsg处于非阻塞模式			•		•
MSG_EOR	接收到记录结束符	•	•	•	•	•
MSG_OOB	接收到带外数据	•	•	•	•	•
MSG_TRUNC	一般数据被截断	•	•	•	•	•

实例：面向连接的客户端

程序清单16-4显示了一个客户端命令，该命令用于与服务器通信以获得系统命令uptime的输出。该服务称为“remote uptime”（简称为“ruptime”）。

程序清单16-4 用于获取服务器uptime的客户端命令

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define MAXADDRLEN  256
```

568

```
#define BUFLEN      128

extern int connect_retry(int, const struct sockaddr *, socklen_t);

void
print_uptime(int sockfd)
{
    int     n;
    char    buf[BUFLEN];

    while ((n = recv(sockfd, buf, BUFLEN, 0)) > 0)
        write(STDOUT_FILENO, buf, n);
    if (n < 0)
        err_sys("recv error");
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
    hint.ai_flags = 0;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_STREAM, 0)) < 0)
            err = errno;
        if (connect_retry(sockfd, aip->ai_addr, aip->ai_addrlen) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd);
            exit(0);
        }
    }
    fprintf(stderr, "can't connect to %s: %s\n", argv[1],
      strerror(err));
    exit(1);
}
```

569

这个程序连接服务器，读取服务器发送过来的字符串并将其打印到标准输出。既然使用SOCK_STREAM套接字，就不能保证在一次recv调用中会读取整个字符串，所以需要重复调用直到返回0。

如果服务器支持多重网络接口或多重网络协议，函数getaddrinfo会返回不止一个候选地址。轮流尝试每个地址，当找到一个允许连接到服务的地址时便可停止。使用程序清单16-2中connect_retry函数来与服务器建立连接。 □

实例：面向连接的服务器

程序清单16-5显示服务器程序，用来提供uptime命令到程序清单16-4的客户端程序的输出。

程序清单16-5 提供系统uptime的服务器程序

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN  128
#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd;
    FILE    *fp;
    char    buf[BUFLEN];

    for (;;) {
        clfd = accept(sockfd, NULL, NULL);
        if (clfd < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "error: %s\n", strerror(errno));
            send(clfd, buf, strlen(buf), 0);
        } else {
            while (fgets(buf, BUFLEN, fp) != NULL)
                send(clfd, buf, strlen(buf), 0);
            pclose(fp);
        }
        close(clfd);
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)  /* best guess */
#endif
```

570

```
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

571

为了找到地址，服务器程序需要获得其运行时的主机名字。一些系统不定义_SC_HOST_NAME_MAX常量，因此这种情况下使用HOST_NAME_MAX。如果系统不定义HOST_NAME_MAX，就自己定义。POSIX.1规定该值的最小值[1]为255字节，不包括终结符，因此定义HOST_NAME_MAX为256以包括终结符。

通过调用gethostname，服务器程序获得主机名字，并查看远程uptime服务地址。可能会有多个地址返回，但简单地选择第一个来建立被动套接字端点。处理多个地址作为练习留给读者。

使用程序清单16-3的initserver函数来初始化套接字端点，在这个端点等待到来的连接请求。（实际上，使用的是程序清单16-9的版本，当在16.6节中讨论套接字选项时，可以了解其中的原因。）□

实例：另一个面向连接的服务器

前面说过采用文件描述符来访问套接字是非常有意义的，因为允许程序对联网环境的网络访问一无所知。程序清单16-6中显示的服务器程序版本显示了这一点。为了代替从uptime命令中读取输出并发送到客户端，服务器安排uptime命令的标准输出和标准错误替换为连接到客户端的套接字端点。

1. POSIX要求主机名长度不小于_POSIX_HOST_NAME_MAX（该值不包括终结符为255，参见本书2.5节说明）。——译者注

程序清单16-6 用于显示命令直接写到套接字的服务器程序

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd, status;
    pid_t   pid;

    for (;;) {
        clfd = accept(sockfd, NULL, NULL);
        if (clfd < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
            exit(1);
        }
        if ((pid = fork()) < 0) {
            syslog(LOG_ERR, "ruptimed: fork error: %s",
              strerror(errno));
            exit(1);
        } else if (pid == 0) {  /* child */
            /*
             * The parent called daemonize (Figure 13.1), so
             * STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO
             * are already open to /dev/null.  Thus, the call to
             * close doesn't need to be protected by checks that
             * clfd isn't already equal to one of these values.
             */
            if (dup2(clfd, STDOUT_FILENO) != STDOUT_FILENO ||
              dup2(clfd, STDERR_FILENO) != STDERR_FILENO) {
                syslog(LOG_ERR, "ruptimed: unexpected error");
                exit(1);
            }
            close(clfd);
            execl("/usr/bin/uptime", "uptime", (char *)0);
            syslog(LOG_ERR, "ruptimed: unexpected return from exec: %s",
              strerror(errno));
        } else {        /* parent */
            close(clfd);
            waitpid(pid, &status, 0);
        }
    }
}

int
main(int argc, char *argv[])
{
```

572

```
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)  /* best guess */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

573

以前的方式是采用popen来运行uptime命令，并从连接到命令标准输出的管道读取输出，现在采用fork来创建一个子进程，并使用dup2使子进程的STDIN_FILENO的副本打开到/dev/null、STDOUT_FILENO和STDERR_FILENO打开到套接字端点。当执行uptime时，命令将结果写到标准输出，该标准输出连到套接字，所以数据被送到ruptime客户端命令。

父进程可以安全地关闭连接到客户端的文件描述符，因为子进程仍旧打开着。父进程等待子进程处理完毕，所以子进程不会变成僵死进程。既然运行uptime花费时间不会太长，父进程在接受下一个连接请求之前，可以等待子进程退出。不过，这种策略不适合子进程运行时间比较长的情况。 □

前面的例子采用面向连接的套接字。但如何选择合适的套接字类型？何时采用面向连接的套接字，何时采用无连接的套接字呢？答案取决于要做的工作以及对错误的容忍程度。

对于无连接套接字，数据包的到来可能已经没有次序，因此当所有的数据不能放在一个包里时，在应用程序里面必须关心包的次序。包的最大尺寸是通信协议的特性。并且对于无连接套接字，包可能丢失。如果应用程序不能容忍这种丢失，必须使用面向连接的套接字。

574

容忍包丢失意味着两个选择。如果想和对方可靠通信，必须对数据包编号，如果发现包丢失，则要求对方重新传输。既然包可能因延迟而疑似丢失，我们要求重传，但该包却又出现，与重传过来的包重复。因此必须识别重复包，如果出现重复包，则将其丢弃。

另外一个选择是通过让用户再次尝试命令来处理错误。对于简单的应用程序，这就足够；但对于复杂的应用程序，这种处理方式通常不是可行的选择，一般在这种情况下使用面向连接的套接字更为可取。

面向连接的套接字的缺陷在于需要更多的时间和工作来建立一个连接，并且每个连接需要从操作系统中消耗更多的资源。

实例：无连接客户端

程序清单16-7中的程序是采用数据报套接字接口的`uptime`客户端命令版本。

程序清单16-7 采用数据报服务的客户端命令

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUFLEN      128
#define TIMEOUT     20

void
sigalrm(int signo)
{
}

void
print_uptime(int sockfd, struct addrinfo *aip)
{
    int     n;
    char    buf[BUFLEN];

    buf[0] = 0;
    if (sendto(sockfd, buf, 1, 0, aip->ai_addr, aip->ai_addrlen) < 0)
        err_sys("sendto error");
    alarm(TIMEOUT);
    if ((n = recvfrom(sockfd, buf, BUFLEN, 0, NULL, NULL)) < 0) {
        if (errno != EINTR)
            alarm(0);
        err_sys("recv error");
    }
    alarm(0);
    write(STDOUT_FILENO, buf, n);
}

int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
    int                 sockfd, err;
    struct sigaction    sa;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
```

575

```
    sa.sa_handler = sigalrm;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) < 0)
        err_sys("sigaction error");
    hint.ai_flags = 0;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));

    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_DGRAM, 0)) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd, aip);
            exit(0);
        }
    }

    fprintf(stderr, "can't contact %s: %s\n", argv[1], strerror(err));
    exit(1);
}
```

除了为SIGALRM增加一个信号处理程序以外，基于数据报的客户端main函数和面向连接的客户端中的类似。使用alarm函数来避免调用recvfrom时无限期阻塞。

对于面向连接的协议，需要在交换数据前连接服务器。对于服务器来说，到来的连接请求已经足够判断出所需提供给客户端的服务。但是对于基于数据报的协议，需要有一种方法来通知服务器需要它提供服务。本例中，只是简单地给服务器发送1字节的消息。服务器接收后从包中得到地址，并使用这个地址来发送响应消息。如果服务器提供多个服务，可以使用这个请求消息来指示所需要的服务，但既然服务器只做一件事情，1字节消息的内容是无关紧要的。

576

如果服务器不在运行状态，客户端调用recvfrom便会无限期阻塞。对于面向连接的例子，如果服务器不运行，connect调用会失败。为了避免无限期阻塞，调用recvfrom之前设置警告时钟。 □

实例：无连接服务器

程序清单16-8中的程序是数据报版本的uptime服务器程序。

程序清单16-8 基于数据报提供系统uptime的服务器程序

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN      128
#define MAXADDRLEN  256
```

```
#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int         n;
    socklen_t   alen;
    FILE        *fp;
    char        buf[BUFLEN];
    char        abuf[MAXADDRLEN];

    for (;;) {
        alen = MAXADDRLEN;
        if ((n = recvfrom(sockfd, buf, BUFLEN, 0,
          (struct sockaddr *)abuf, &alen)) < 0) {
            syslog(LOG_ERR, "ruptimed: recvfrom error: %s",
              strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "error: %s\n", strerror(errno));
            sendto(sockfd, buf, strlen(buf), 0,
              (struct sockaddr *)abuf, alen);
        } else {
            if (fgets(buf, BUFLEN, fp) != NULL)
                sendto(sockfd, buf, strlen(buf), 0,
                  (struct sockaddr *)abuf, alen);
            pclose(fp);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)  /* best guess */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_protocol = 0;
```

577

```
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_DGRAM, aip->ai_addr,
          aip->ai_addrlen, 0)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

578

服务器程序在recvfrom中阻塞等待服务请求。当一个请求到达时，保存请求者地址并使用popen来运行uptime命令。采用sendto函数将输出发送到客户端，其目标地址就设为刚才的请求者地址。□

16.6 套接字选项

套接字机制提供两个套接字选项接口来控制套接字行为。一个接口用来设置选项，另一个接口允许查询一个选项的状态。可以获取或设置三种选项：

(1) 通用选项，工作在所有套接字类型上。

(2) 在套接字层次管理的选项，但是依赖于下层协议的支持。

(3) 特定于某协议的选项，为每个协议所独有。

Single UNIX Specification仅定义了套接字层的选项（上述三种选项中的前两种选项）。

可以采用setsockopt函数来设置套接字选项。

```
#include <sys/socket.h>

int setsockopt(int sockfd, int level, int option, const void *val,
               socklen_t len);
```

返回值：若成功则返回0，若出错则返回-1

参数*level*标识了选项应用的协议。如果选项是通用的套接字层选项，*level*设置成SOL_SOCKET。否则，*level*设置成控制这个选项的协议号。例如，对于TCP选项，这是IPPROTO_TCP，对于IP选项，这是IPPROTO_IP。表16-10总结了Single UNIX Specification所定义的通用套接字层的选项。

参数*val*根据选项的不同指向一个数据结构或者一个整数。一些选项是on/off开关。如果整数非零，那么选项被启用。如果整数为零，那么该选项被禁止。参数*len*指定了*val*指向的对象的大小。

可以使用getsockopt函数来发现选项的当前值。

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int option, void *restrict val,
               socklen_t *restrict lenp);
```

返回值：若成功则返回0，若出错则返回-1

表16-10 套接字选项

选 项	参数val类型	描 述
SO_ACCEPTCONN	int	返回信息指示该套接字是否能监听（仅getsockopt）
SO_BROADCAST	int	如果*val非零，广播数据包
SO_DEBUG	int	如果*val非零，启用网络驱动调试功能
SO_DONTROUTE	int	如果*val非零，绕过通常路由
SO_ERROR	int	返回挂起的套接字错误并清除（仅getsockopt）
SO_KEEPALIVE	int	如果*val非零，启用周期性keep-alive消息
SO_LINGER	struct linger	当有未发消息并且套接字关闭时，延迟时间
SO_OOBINLINE	int	如果*val非零，将带外数据放在普通数据中
SO_RCVBUF	int	以字节为单位的接收缓冲区大小
SO_RCVLOWAT	int	接收调用中返回的以字节为单位的最小数据量
SO_RCVTIMEO	struct timeval	套接字接收调用的超时值
SO_REUSEADDR	int	如果*val非零，重用bind中的地址
SO_SNDBUF	int	以字节为单位的发送缓冲区大小
SO_SNDLOWAT	int	发送调用中以字节为单位的发送的最小数据量
SO_SNDTIMEO	struct timeval	套接字发送调用的超时值
SO_TYPE	int	标识套接字类型（仅getsockopt）

注意到参数*lenp*是一个指向整数的指针。在调用getsockopt之前，设置该整数为复制选项缓冲区的大小。如果实际的尺寸大于此值，选项会被截断而不报错；如果实际尺寸正好等于或者小于此值，那么返回时将此值更新为实际尺寸。 579

实例

当服务器终止并尝试立即重启时，程序清单16-3中的函数不会正常工作。除非超时（这个通常约为几分钟），通常TCP的实现不允许绑定同一个地址。幸运的是套接字选项SO_REUSEADDR允许越过这个限制，如程序清单16-9所示。

程序清单16-9 采用地址复用初始化套接字端点

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd, err;
    int reuse = 1;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &reuse,
      sizeof(int)) < 0) {
        err = errno;
```

580

```
            goto errout;
        }
        if (bind(fd, addr, alen) < 0) {
            err = errno;
            goto errout;
        }
        if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
            if (listen(fd, qlen) < 0) {
                err = errno;
                goto errout;
            }
        }
        return(fd);

errout:
    close(fd);
    errno = err;
    return(-1);
}
```

为了启用SO_REUSEADDR选项，在setsockopt中*val*的参数设置为一个非零整数的地址。设置*len*参数为*val*所指的对象的大小。□

16.7 带外数据

带外数据（Out-of-band data）是一些通信协议所支持的可选特征，允许更高优先级的数据比普通数据优先传输。即使传输队列已经有数据，带外数据先行传输。TCP支持带外数据，但是UDP不支持。套接字接口对带外数据的支持，很大程度受TCP带外数据具体实现的影响。

TCP将带外数据称为“紧急”数据（“urgent”data）。TCP仅支持一个字节的紧急数据，但是允许紧急数据在普通数据传递机制数据流之外传输。为了产生紧急数据，在三个send函数中任何一个指定标志MSG_OOB。如果带MSG_OOB标志传输字节超过一个时，最后一个字节被看作紧急数据字节。

如果安排发生套接字信号，当接收到紧急数据时，那么发送信号SIGURG。在3.14节和14.6.2节中，可以看到在fcntl中使用F_SETOWN命令来设置一个套接字的所有权。如果fcntl中第三个参数为正值，那么指定了进程ID；如果为非-1的负值，那么代表了进程组ID。因此，通过调用以下函数，可以安排进程接收一个套接字的信号。

581

```
fcntl(sockfd, F_SETOWN, pid);
```

F_GETOWN命令可以用来获得当前套接字所有权。对于F_SETOWN命令，一个负值代表一个进程组ID，一个正值代表进程ID。因此，调用

```
owner = fcntl(sockfd, F_GETOWN, 0);
```

返回值owner，如果owner为正值，则owner等于配置为接受套接字信号的进程ID；如果owner为负值，则其绝对值为接受套接字信号的进程组ID。

TCP支持*紧急标记*（urgent mark）的概念：在普通数据流中紧急数据所在的位置。如果采用套接字选项SO_OOBINLINE，那么可以在普通数据中接收紧急数据。为帮助判断是否接收到紧急标记，可以使用函数sockatmark。

```
#include <sys/socket.h>

int sockatmark(int sockfd);
```

返回值：若在标记处则返回1，若没有在标记处则返回0，若出错则返回-1

当下一个要读的字节在紧急标志所标识的位置时，sockatmark返回1。

当带外数据出现在套接字读取队列时，select函数（14.5.1节）会返回一个文件描述符并且拥有一个异常状态挂起。可以在普通数据流上接受紧急数据，或者在某个recv函数中采用MSG_OOB标志在其他队列数据之前接收紧急数据。TCP队列仅有一字节的紧急数据，如果在接收当前的紧急数据字节之前又有新的紧急数据到来，那么当前的字节会被丢弃。

16.8 非阻塞和异步I/O

通常，recv函数没有数据可用时会阻塞等待。同样地，当套接字输出队列没有足够空间来发送消息时函数send会阻塞。在套接字非阻塞模式下，行为会改变。在这种情况下，这些函数不会阻塞而是失败，设置errno为EWOULDBLOCK或者EAGAIN。当这些发生时，可以使用poll或select来判断何时能接收或者传输数据。

在Single UNIX Specification中，其实时扩展包含对通用异步I/O机制的支持。套接字机制有自己的方式来处理异步I/O，但是在Single UNIX Specification中没有标准化。一些文献把经典的基于套接字的异步I/O机制称为“基于信号的I/O”，以区别于实时扩展中的异步I/O机制。 582

在基于套接字的异步I/O中，当能够从套接字中读取数据，或者套接字写队列中的空间变得可用时，可以安排发送信号SIGIO。通过两个步骤来使用异步I/O：

(1) 建立套接字拥有者关系，信号可以被传送到合适的进程。

(2) 通知套接字当I/O操作不会阻塞时发信号告知。

可以使用三种方式来完成第一个步骤：

(1) 在fcntl使用F_SETOWN命令。

(2) 在ioctl中使用FIOSETOWN命令。

(3) 在ioctl中使用SIOCSPGRP命令。

要完成第二个步骤，有两个选择：

(1) 在fcntl中使用F_SETFL命令并且启用文件标志O_ASYNC。

(2) 在ioctl中使用FIOASYNC。

虽然有不少选择，但不是普遍得到支持。表16-11总结了本书讨论平台对这些选项的支持情况。表中以 • 显示提供支持，以+显示支持依赖于特定的域。例如，在Linux上，UNIX域套接字不支持FIOSETOWN和SIOCSPGRP。

表16-11 异步套接字I/O管理命令

机 制	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
fcntl(fd, F_SETOWN, pid)	•	•	•	•	•
ioctl(fd, FIOSETOWN, pid)		•	+	•	•
ioctl(fd, SIOCSPGRP, pid)		•	+	•	•
fcntl(fd, F_SETFL, flags|O_ASYNC)		•	•	•	
ioctl(fd, FIOASYNC, &n);		•	•	•	•

16.9　小结

在本章中，考查了IPC机制，这种机制允许一个进程与另外一个进程通信，无论是不同的机器上还是同一机器中。讨论了套接字端点如何命名，在连接服务器时，如何发现所要用的地址。

我们给出了采用无连接的套接字（例如，基于数据报）和面向连接套接字的客户端和服务器的例子。简要讨论了异步和非阻塞的套接字I/O，以及用于管理套接字选项的接口。

在下一章，将会考查一些高级IPC主题，包括在同一台机器上如何使用套接字传送文件描述符。

583

习题

16.1　写一个程序判断系统的字节序。

16.2　写一个程序，在至少两种不同的平台上打印出所支持套接字的stat结构成员，并且描述这些结果不同之处。

16.3　程序清单16-5中的程序提供仅单一端点的服务。修改这个程序，使其同时支持多个端点的服务（每个具有不同的地址）。

16.4　写一个客户端程序和一个服务端程序，返回指定主机上当前运行的进程数量。

16.5　在程序清单16-6的程序中，服务器等待子进程执行uptime命令，子进程完成后退出，服务器才接受下一个连接请求。重新设计服务器，使得为一个请求服务时并不耽误处理到来的连接请求。

16.6　写两个库例程：一个在套接字上允许异步I/O，一个在套接字禁止异步I/O。使用表16-11来保证函数能够在所有平台上运行，并且支持尽可能多的套接字类型。

584

第17章

高级进程间通信

17.1 引言

前面两章讨论了UNIX系统提供的各种IPC，包括管道和套接字。本章介绍两种高级IPC：基于STREAMS的管道（STREAMS-based pipe）以及UNIX域套接字（UNIX domain socket），并说明它们的应用方法。使用这些IPC，可以在进程间传送打开文件描述符。服务进程可以使它们的打开文件描述符与指定的名字相关联，客户进程可以使用这些名字与服务进程通信。我们会了解到操作系统如何为每一个客户进程提供一个独用的IPC通道。构成本章所述技术基础的很多思想来自于Pressotto和Ritchie[1990]的论文。

17.2 基于STREAMS的管道

基于STREAMS的管道（简称为STREAMS管道，STREAMS pipe）是一个双向（全双工）管道。单个STREAMS管道就能向父、子进程提供双向的数据流。

回忆15.1节，Solaris支持STREAMS管道，Linux的可选附加包也提供了STREAMS管道。

图17-1显示了观察STREAMS管道的两种方式。它与图15-1的唯一区别是双向箭头连线，因为STREAMS管道是全双工的，数据可以双向流动。 585

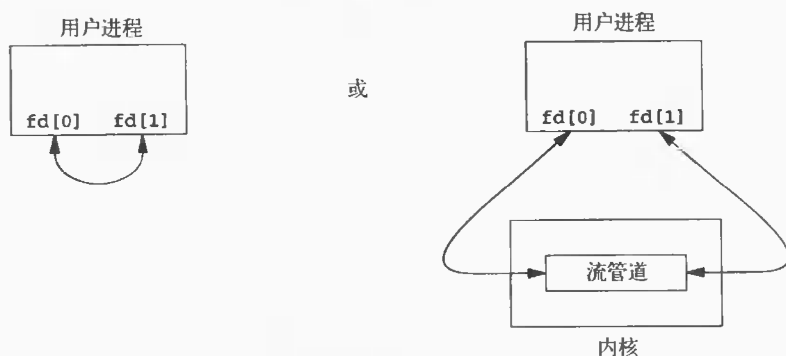


图17-1 观察STREAMS管道的两种方式

如若从内部观察STREAMS管道（图17-2），可以看到它简单得只包含两个流首，每个流首的写队列（WQ）指向另一个流首的读队列（RQ），写入管道一端的数据被放入另一端的读队列的消息中。

因为STREAMS管道是一个流，所以可将STREAMS模块压入到该管道的任一端（图17-3）。

但是，如果我们在一端压入了一个模块，那么并不能在另一端弹出该模块。如果想要删除它，则必须从原压入端删除。

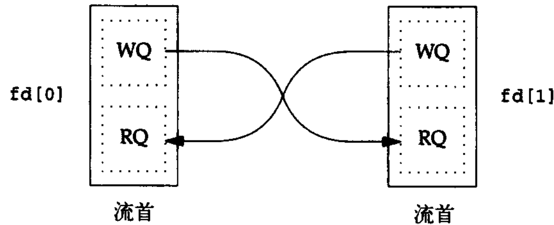
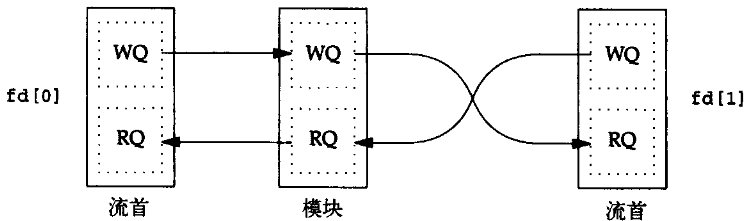


图17-2　STREAMS管道的内部结构



586

图17-3　带模块的STREAMS管道内部结构

假定不做类似于压入模块这样复杂的处理，那么，除了支持在streamio(7)中描述的大多数流ioctl命令外，STREAMS管道的运行行为与非STREAMS管道并无差别。在17.2.2节，我们将见到一个实例，它将一个流模块压入STREAMS管道中，当我们在文件系统中给管道一个名字时，它提供了唯一连接。

下面用一个STREAMS管道再次实现程序清单15-9中的协同进程实例。程序清单17-1是新的main函数。add2协同进程与程序清单15-8中的相同。本程序调用了创建单个STREAMS管道的新函数s_pipe。(下面紧接着将说明该函数的STREAMS管道和UNIX域套接字版本。)

程序清单17-1　用STREAMS管道驱动add2过滤进程的程序

```
#include "apue.h"

static void sig_pipe(int);      /* our signal handler */

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (s_pipe(fd) < 0)         /* need only a single stream pipe */
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
```

```
    } else if (pid > 0) {                          /* parent */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }

        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                    /* child */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO &&
          dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        if (fd[1] != STDOUT_FILENO &&
          dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

587

父进程只使用fd[0]，子进程只使用fd[1]。因为STREAMS管道的每一端都是全双工的，所以父进程读、写fd[0]，而子进程将fd[1]复制到标准输入和标准输出。图17-4显示了由此构成的各描述符。注意，除了STREAMS管道的全双工性质外，该实例并没有使用STREAMS管道的其他特性，所以如果使用不是基于STREAMS的全双工管道，它同样能行。

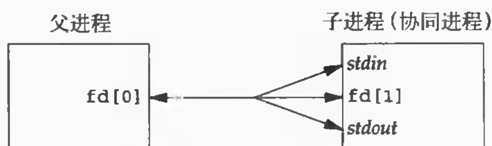


图17-4 为协同进程所作的描述符安排

Rago [1993]较详细地描述了基于STREAMS的管道。回忆表15-1，FreeBSD 支持全双工管道，但这些管道并不是基于STREAMS 机制的。

s_pipe函数定义为与标准pipe函数类似。它的调用参数与pipe相同，但返回的描述符以读-写模式打开。

588

实例：基于STREAMS的s_pipe函数

程序清单17-2是基于STREAMS的s_pipe函数版本。它只是简单地调用创建全双工管道的标准pipe函数。

程序清单17-2　基于STREAMS的s_pipe函数版本

```
#include "apue.h"

/*
 * Returns a STREAMS-based pipe, with the two file descriptors
 * returned in fd[0] and fd[1].
 */
int
s_pipe(int fd[2])
{
    return(pipe(fd));
}
```

17.2.1　命名的STREAMS管道

通常，管道仅在相关进程之间使用：子进程继承父进程的管道。在15.5节，我们见到无关进程可以使用FIFO进行通信，但是这仅仅提供单向通信。STREAMS机制提供了一种途径，使得进程可以给予管道一个文件系统中的名字。这就避免了单向FIFO的问题。

我们可以用fattach函数给STREAMS管道一个文件系统中的名字。

```
#include <stropts.h>

int fattach(int filedes, const char *path);
```

返回值：若成功则返回0，若出错则返回-1

*path*参数必须引用一个现存的文件，调用进程应当或者拥有该文件并且对它具有写权限，或者正在以超级用户特权运行。

一旦STREAMS管道连接到文件系统名字空间，那么原来使用该名字的底层文件就不再是可访问的。打开该名字的任一进程将能访问相应管道，而不是访问原先的文件。在调用fattach之前打开底层文件的任一进程可以继续访问该文件。确实，一般而言，这些进程并不知道该名字现在引用了另外一个文件。

图17-5显示了连接到路径名/tmp/pipe的一条管道。只有管道的一端连接到文件系统中一个名字上。另一端用来与打开该连接文件名的进程通信。虽然fattach函数可将任何种类的STREAMS文件描述符与文件系统中的一个名字相连接，但它最主要用于将一个名字给予一STREAMS管道。

589

一个进程可以调用fdetach函数撤销STREAMS管道文件与文件系统中名字的关联关系。

```
#include <stropts.h>

int fdetach(const char *path);
```

返回值：若成功则返回0，若出错则返回-1

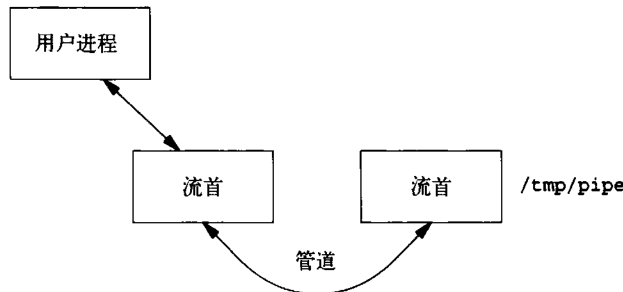


图17-5 一条管道安装到文件系统的一个名字上

在调用fdetach函数之后，先前依靠打开path而能访问STREAMS管道的进程仍可继续访问该管道，但是在此之后打开*path*的进程将访问驻留在文件系统中的底层文件。

17.2.2 唯一连接

虽然我们可以将STREAMS管道的一端连接到文件系统的名字空间，但是如果多个进程都想要用命名STREAMS管道与服务器进程通信，那么仍然存在问题。若几个客户进程同时将数据写至一管道，那么这些数据就会混合交错。即使我们保证客户进程写的字节数小于PIPE_BUF，使得写操作是原子性的，但是仍无法保证服务器进程将数据送回所期望的某个客户进程，也无法保证该客户进程一定会读此消息。当多个客户进程同时读一管道时，我们无法调度具体哪一个客户进程去读我们所发送的消息。

connld STREAMS模块解决了这一问题。在将一个STREAMS管道连接到文件系统的一个名字之前，服务器进程可将connld模块压入要被连接管道的一端。其结果示于图17-6。

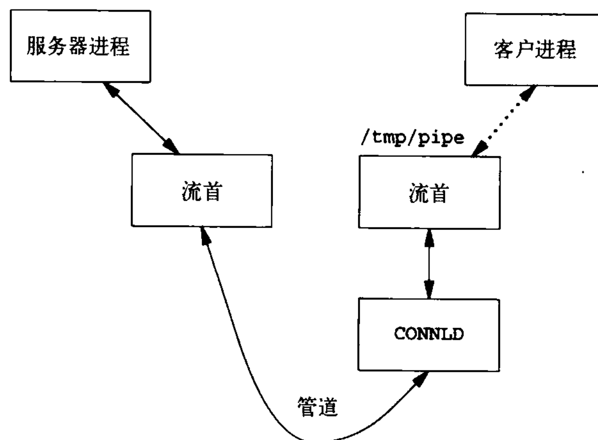


图17-6 为唯一连接设置connld

在图17-6中，服务器进程已将管道的一端连接至/tmp/pipe。我们用虚线指示客户进程正在打开所连接的STREAMS管道。一旦打开操作完成，则服务器进程、客户进程和STREAMS管道之间的关系示于图17-7中。

590

客户进程决不会接收到它所打开管道端的打开文件描述符。作为替代，操作系统创建了一个新管道，对客户进程返回其一端，作为它打开/tmp/pipe的结果。系统将此新管道另一端的文件描述符经由已存在的连接管道发送给服务器进程，结果在客户进程和服务器进程之间构

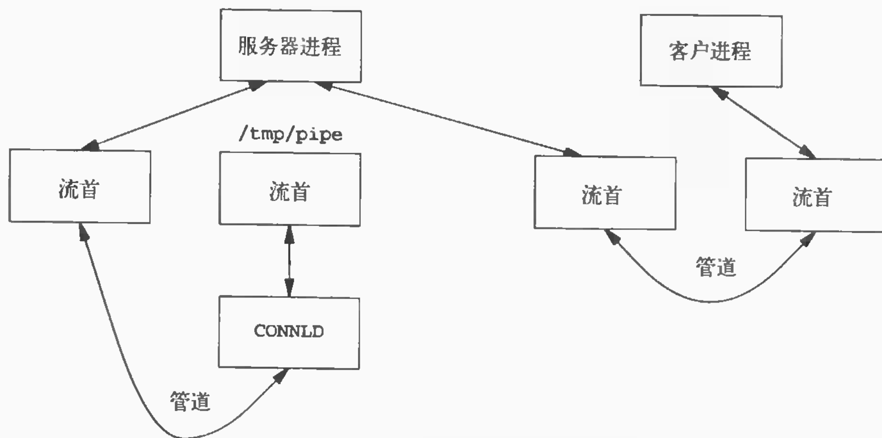


图17-7　用connld构造唯一连接

[591] 成了唯一连接。我们将在17.4.1中了解到用STREAMS管道传送文件描述符的机制。

> fattach函数是在mount系统调用之上构造的。这种设施被认为是安装流（mounted stream）。安装流和connld模块是由Presotto和Ritchie[1990]为Research UNIX系统开发的。然后，SVR4采用了这些机制。

现在，我们将开发三个函数，使用这些函数可以创建在无关进程之间的唯一连接。这些函数模仿了在16.4节中讨论过的面向连接的套接字函数。在此处，我们使用STREAMS管道作为底层通信机制，在17.3节我们则将见到用UNIX域套接字实现的同样这三个函数。

```
#include "apue.h"

int serv_listen(const char *name);
```

返回值：若成功则返回要侦听的文件描述符，若出错则返回负值

```
int serv_accept(int listenfd, uid_t *uidptr);
```

返回值：若成功则返回新文件描述符，若出错则返回负值

```
int cli_conn(const char *name);
```

返回值：若成功则返回文件描述符，若出错则返回负值

服务器进程调用serv_listen函数（程序清单17-3）声明它要在一个众所周知的名字（文件系统中的某个路径名）上侦听客户进程的连接请求。当客户进程想要连接至服务器进程时，它们将使用该名字。serv_listen函数的返回值是STREAMS管道的服务器进程端。

程序清单17-3　使用STREAMS管道的serv_listen函数

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>

/* pipe permissions: user rw, group rw, others rw */
#define FIFO_MODE  (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*
 * Establish an endpoint to listen for connect requests.
 * Returns fd if all OK, <0 on error
```

```
 */
int
serv_listen(const char *name)
{
    int     tempfd;
    int     fd[2];

    /*
     * Create a file: mount point for fattach().
     */
    unlink(name);
    if ((tempfd = creat(name, FIFO_MODE)) < 0)
        return(-1);
    if (close(tempfd) < 0)
        return(-2);
    if (pipe(fd) < 0)
        return(-3);
    /*
     * Push connld & fattach() on fd[1].
     */
    if (ioctl(fd[1], I_PUSH, "connld") < 0) {
        close(fd[0]);
        close(fd[1]);
        return(-4);
    }
    if (fattach(fd[1], name) < 0) {
        close(fd[0]);
        close(fd[1]);
        return(-5);
    }
    close(fd[1]);   /* fattach holds this end open */

    return(fd[0]);  /* fd[0] is where client connections arrive */
}
```

592

服务器进程使用serv_accept函数（程序清单17-4）等待客户进程连接请求的到达。当一个请求到达时，系统自动创建一个新的STREAMS管道，serv_accept函数向服务器进程返回该STREAMS管道的一端。另外，客户进程的有效用户ID存放在*uidptr*指向的存储区中。

程序清单17-4 使用STREAMS管道的serv_accept函数

```
#include "apue.h"
#include <stropts.h>

/*
 * Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID.
 * Returns new fd if all OK, <0 on error.
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    struct strrecvfd    recvfd;

    if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
        return(-1);     /* could be EINTR if signal caught */
    if (uidptr != NULL)
        *uidptr = recvfd.uid;   /* effective uid of caller */
    return(recvfd.fd);  /* return the new descriptor */
}
```

客户进程调用cli_conn函数（程序清单17-5）连接至服务器进程。客户进程指定的参数*name*必须与服务器进程调用serv_listen函数时所用的相同。函数返回时，客户进程得到接连至服务器进程的文件描述符。

593

程序清单17-5　用STREAMS管道的cli_conn函数

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>

/*
 * Create a client endpoint and connect to a server.
 * Returns fd if all OK, <0 on error.
 */
int
cli_conn(const char *name)
{
    int     fd;

    /* open the mounted stream */
    if ((fd = open(name, O_RDWR)) < 0)
        return(-1);
    if (isastream(fd) == 0) {
        close(fd);
        return(-2);
    }
    return(fd);
}
```

我们对返回的描述符是否引用STREAMS设备进行了二次检验，以防服务器进程没被启动而路径名仍存在于文件系统中。在17.6节中，我们将会了解到如何使用这三个函数。

17.3　UNIX域套接字

UNIX域套接字用于在同一台机器上运行的进程之间的通信。虽然因特网域套接字可用于同一目的，但UNIX域套接字的效率更高。UNIX域套接字仅仅复制数据；它们并不执行协议处理，不需要添加或删除网络报头，无需计算检验和，不要产生顺序号，无需发送确认报文。

UNIX域套接字提供流和数据报两种接口。UNIX域数据报服务是可靠的，既不会丢失消息也不会传递出错。UNIX域套接字是套接字和管道之间的混合物。为了创建一对非命名的、相互连接的UNIX域套接字，用户可以使用它们面向网络的域套接字接口，也可使用socketpair函数。

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

返回值：若成功则返回0，若出错则返回−1

594

虽然该接口具有足够的一般性，socketpair可用于任意域，但操作系统通常仅对UNIX域提供支持。

实例：使用UNIX域套接字的s_pipe函数

程序清单17-6是基于套接字的s_pipe函数版本，该函数曾出现于程序清单17-2。s_pipe

函数创建一对相连接的UNIX域流套接字。

程序清单17-6 s_pipe函数的套接字版本

```
#include "apue.h"
#include <sys/socket.h>

/*
 * Returns a full-duplex "stream" pipe (a UNIX domain socket)
 * with the two file descriptors returned in fd[0] and fd[1].
 */
int
s_pipe(int fd[2])
{
    return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}
```

某些基于BSD的系统使用UNIX域套接字实现管道。但当调用pipe时，第一描述符的写端和第二描述符的读端都被关闭。为了得到全双工管道，我们必须直接调用socketpair。

□

17.3.1 命名UNIX域套接字

虽然socketpair函数创建相互连接的一对套接字，但是每一个套接字都没有名字。这意味着无关进程不能使用它们。

在16.3.4节，我们学习了如何将一个地址绑定一因特网域套接字。恰如因特网域套接字一样，我们也可以命名UNIX域套接字，并可将其用于告示服务。但是要注意的是，UNIX域套接字使用的地址格式不同于因特网域套接字。

回忆16.3节，套接字地址格式可能随实现而变。UNIX域套接字的地址由sockaddr_un结构表示。在Linux 2.4.22和Solaris 9中，sockaddr_un结构按下列形式定义在头文件<sys/un.h>中：

```
struct sockaddr_un {
    sa_family_t  sun_family;     /* AF_UNIX */
    char         sun_path[108];  /* pathname */
};
```


但是在FreeBSD 5.2.1和Mac OS X 10.3中，sockaddr_un结构定义如下：

```
struct sockaddr_un {
    unsigned char  sun_len;        /* length including null */
    sa_family_t    sun_family;     /* AF_UNIX */
    char           sun_path[104];  /* pathname */
};
```

sockaddr_un结构的sun_path成员包含一路径名。当我们将一地址绑定至UNIX域套接字时，系统用该路径名创建一类型为S_IFSOCK的文件。

该文件仅用于向客户进程告知套接字名字。该文件不能打开，也不能由应用程序用于通信。

如果当我们试图绑定地址时，该文件已经存在，那么bind请求失败。当关闭套接字时，并不自动删除该文件，所以我们必须确保在应用程序终止前，对该文件执行解除链接操作。

实 例

程序清单17-7是一个例子，它将一地址绑定一UNIX域套接字。

程序清单17-7　将一个地址绑定一UNIX域套接字

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int
main(void)
{
    int fd, size;
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket failed");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("bind failed");
    printf("UNIX domain socket bound\n");
    exit(0);
}
```

当运行此程序时，bind请求成功执行，但是如若第二次运行该程序，则出错返回，其原因是该文件已经存在。在删除该文件之前，程序清单17-7不会成功运行。

596

```
$ ./a.out                                              运行该程序
UNIX domain socket bound
$ ls -l foo.socket                                     查看套接字文件
srwxrwxr-x  1 sar       0 Aug 22 12:43 foo.socket
$ ./a.out                                              试图再次运行该程序
bind failed: Address already in use
$ rm foo.socket                                        删除该套接字文件
$ ./a.out                                              第3次运行该程序
UNIX domain socket bound                               现在成功啦
```

确定绑定地址长度的方法是，先确定sun_path成员在sockaddr_un结构中的偏移量，然后将此与路径名长度（不包括终止null字符）相加。因为在sun_path之前的成员与实现相关，所以我们使用<stddef.h>头文件（apue.h中包括）中的offsetof宏计算sun_path成员从结构开始处的偏移量。如果查看<stddef.h>，则可见到类似于下列形式的定义：

```
#define offsetof(TYPE, MEMBER)   ((int)&((TYPE *)0)->MEMBER)
```

假定该结构从地址0开始，此表达式求得成员起始地址的整型值。　□

17.3.2 唯一连接

服务器进程可以使用标准bind、listen和accept函数，为客户进程安排一个唯一UNIX域连接（unique UNIX domain connection）。客户进程使用connect与服务器进程联系；服务器进程接受了connect请求后，在服务器进程和客户进程之间就存在了唯一连接。这种风格的操作与我们在程序清单16-4和程序清单16-5中所示的对因特网域套接字的操作相同。

程序清单17-8示出了serv_listen函数的UNIX域套接字版本。

程序清单17-8　UNIX域套接字的serv_listen函数

```
#include "apue.h"
#include <sys/socket.h>
```

```
#include <sys/un.h>
#include <errno.h>

#define QLEN    10

/*
 * Create a server endpoint of a connection.
 * Returns fd if all OK, <0 on error.
 */
int
serv_listen(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un;

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);
    unlink(name);   /* in case it already exists */

    /* fill in socket address structure */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }

    if (listen(fd, QLEN) < 0) { /* tell kernel we're a server */
        rval = -3;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
```

597

首先，我们调用socket创建一个UNIX域套接字。然后将欲赋予套接字的众所周知路径名填入sockaddr_un结构。该结构是调用bind的参数。注意，我们不需要设置某些平台提供的sun_len字段，操作系统用传送给bind函数的地址长度设置该字段。

最后，调用listen函数（16.4节）以通知内核该进程将作为服务器进程等待客户进程的连接请求。当收到一个客户进程的连接请求后，服务器进程调用serv_accept函数（程序清单17-9）。

程序清单17-9 UNIX域套接字的serv_accept函数

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>
#include <errno.h>

#define STALE   30  /* client's name can't be older than this (sec) */
```

```
/*
 * Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID from the pathname
 * that it must bind before calling us.
 * Returns new fd if all OK, <0 on error
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    int                 clifd, len, err, rval;
    time_t              staletime;
    struct sockaddr_un  un;
    struct stat         statbuf;

    len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0)
        return(-1);     /* often errno=EINTR, if signal caught */

    /* obtain the client's uid from its calling address */
    len -= offsetof(struct sockaddr_un, sun_path); /* len of pathname */
    un.sun_path[len] = 0;           /* null terminate */

    if (stat(un.sun_path, &statbuf) < 0) {
        rval = -2;
        goto errout;
    }
#ifdef  S_ISSOCK    /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0) {
        rval = -3;      /* not a socket */
        goto errout;
    }
#endif
    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
          rval = -4;    /* is not rwx------ */
          goto errout;
    }

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
        statbuf.st_ctime < staletime ||
        statbuf.st_mtime < staletime) {
          rval = -5;    /* i-node is too old */
          goto errout;
    }

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid;   /* return uid of caller */
    unlink(un.sun_path);        /* we're done with pathname now */
    return(clifd);

errout:
    err = errno;
    close(clifd);
    errno = err;
    return(rval);
}
```

服务器进程在调用serv_accept中阻塞以等待一客户进程调用cli_conn。从accept返回时，返回值是连接到客户进程的崭新的描述符。(这有些类似于connld模块对于STREAMS子系统所做的那样。）另外，accept函数也经由其第二个参数（指向sockaddr_un结构的指

针）返回客户进程赋予其套接字的路径名（包含客户进程ID的名字）。接着，程序在此路径名结尾处填补null字符，然后调用stat函数。这使我们验证该路径名确实是一个套接字，其权限仅允许用户-读、用户-写以及用户-执行。我们也验证与套接字相关联的3个时间不比当前时间早30秒。（回忆6.10节，time函数返回当前时间和日期，单位是秒，起始点是公元1970年1月1日00.00:00。）

如若通过了所有这些检验，则可认为客户进程的身份（其有效用户ID）是该套接字的所有者。虽然这种检验并不完善，但这是对当前系统所能做到的最佳方案。（如若内核能像处理I_RECVFD ioctl命令那样，使accept返回有效用户ID，则会更好一些。）

客户进程调用cli_conn函数（程序清单17-10）对联向服务器进程的连接进行初始化。

程序清单17-10 用于UNIX域套接字的cli_conn函数

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define CLI_PATH    "/var/tmp/"     /* +5 for pid = 14 chars */
#define CLI_PERM    S_IRWXU         /* rwx for user only */

/*
 * Create a client endpoint and connect to a server.
 * Returns fd if all OK, <0 on error.
 */
int
cli_conn(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un;

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /* fill socket address structure with our address */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    sprintf(un.sun_path, "%s%05d", CLI_PATH, getpid());
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);

    unlink(un.sun_path);        /* in case it already exists */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }
    if (chmod(un.sun_path, CLI_PERM) < 0) {
        rval = -3;
        goto errout;
    }

    /* fill socket address structure with server's address */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);
    if (connect(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -4;
        goto errout;
    }
```

600

```
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
```

我们调用socket函数创建UNIX域套接字的客户进程端，然后用客户进程专有的名字填入sockaddr_un结构。

我们不让系统为我们选择一个默认地址，原因是这样处理后，服务器进程将不能区分各个客户进程。于是，我们绑定我们自己的地址，在开发使用套接字的客户端程序时通常并不采用这一步骤。

我们绑定的路径名的最后5个字符来自客户进程ID。我们调用unlink，以防该路径名已经存在。然后，调用bind将名字赋予客户进程套接字。这在文件系统中创建了一个套接字文件，所用的名字与被绑定路径名一样。接着，调用chmod关闭除用户-读、用户-写以及用户-执行以外的其他权限。在serv_accept中，服务器进程检验这些权限以及套接字用户ID以验证客户进程的身份。

然后，我们必须填充另一个sockaddr_un结构，这次用的是服务器进程众所周知的路径名。最后，调用connect函数初始化与服务器进程的连接。

17.4 传送文件描述符

在进程间传送打开的文件描述符的能力是非常有用的，可以用它对客户进程/服务器进程应用进行不同的设计。它使一个进程（一般是服务器进程）能够处理为打开一个文件所要求的一切操作（具体如将网络名翻译为网络地址、拨号调制解调器、协商文件锁等）以及向调用进程送回一描述符，该描述符可被用于以后的所有I/O函数。涉及打开文件或设备的所有细节对客户进程而言都是隐藏了的。

601

下面进一步说明从一个进程向另一个进程“传送一打开的文件描述符”的含义。回忆图3-2，其中显示了两个进程，它们打开了同一文件。虽然它们共享同一v节点表，但每个进程都有它自己的文件表项。

当一个进程向另一个进程传送一打开的文件描述符时，我们想要发送进程和接收进程共享同一文件表项。图17-8显示了所希望的安排。

在技术上，发送进程实际上向接收进程传送一个指向一打开文件表项的指针。该指针被分配存放在接收进程的第一个可用描述符项中。（注意，不要造成错觉，以为发送进程和接收进程中的描述符编号是相同的，通常它们是不同的。）两个进程共享同一打开文件表项，在这一点上与fork之后，父、子进程共享打开文件表项的情况完全相同（参见图8-1）。

当发送进程将描述符传送给接收进程后，通常它关闭该描述符。发送进程关闭该描述符并不造成关闭该文件或设备，其原因是该描述符对应的文件仍被视为由接收进程打开（即使接收进程尚未接收到该描述符）。

602

下面定义本章使用的三个函数以发送和接收文件描述符。本节将会给出对于STREAMS和套接字的这三个函数的不同实现代码。

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);

int send_err(int fd, int status, const char *errmsg);
```

两个函数返回值：若成功则返回0，若出错则返回−1

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

返回值：若成功则返回文件描述符，若出错则返回负值

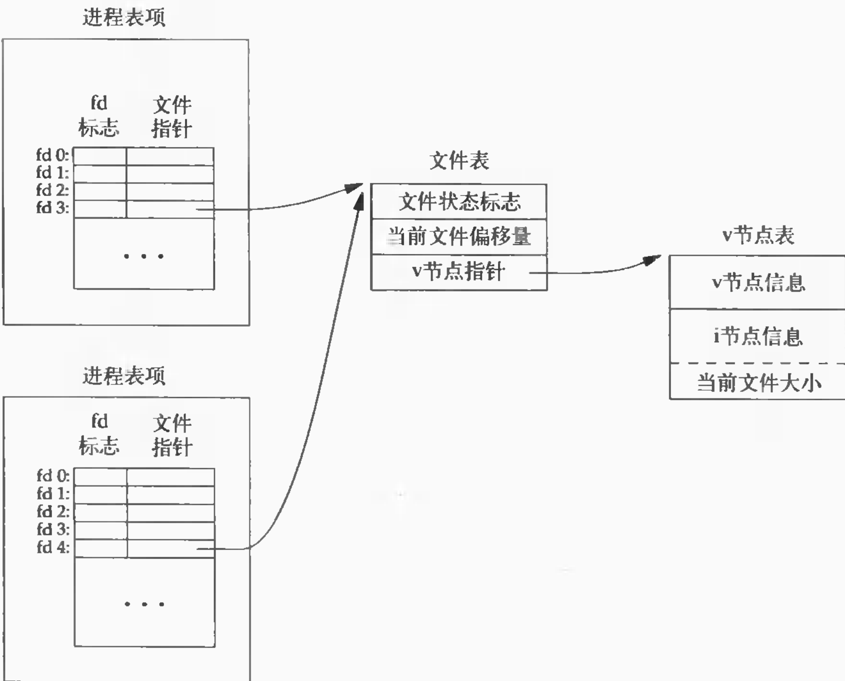


图17-8 从顶部进程传送一个打开的文件至底部进程

当一个进程（通常是服务器进程）希望将一个描述符传送给另一个进程时，它调用send_fd或send_err。等待接收描述符的进程（客户进程）调用recv_fd。

send_fd经由*fd*代表的STREAMS管道或UNIX域套接字发送描述符*fd_to_send*。

> 我们将使用术语s管道表示可实现为STREAMS管道或UNIX域流套接字的双向通信通道。

send_err函数用*fd*发送*errmsg*以及后随的*status*字节。*status*的值应在−1到−255之间。

客户进程调用recv_fd接收一描述符。如果一切正常（发送者调用了send_fd），则作为函数值返回非负描述符。否则，返回值是由send_err发送的*status*（−1到−255之间的一个值）。另外，如果服务器进程发送了一条出错消息，则客户进程调用它自己的*userfunc*处理该消息。*userfunc*的第一个参数是常量STDERR_FILENO，然后是指向出错消息的指针及其长度。*userfunc*函数的返回值是已写的字节数或负的出错编号值。客户进程常将*userfunc*指定为通常的write函数。

我们实现了用于这三个函数的我们自己制定的协议。为发送一描述符，send_fd先发送两个0字节，然后是实际描述符。为了发送一条出错消息，send_err发送*errmsg*，然后是1个0字

节，最后是*status*字节的绝对值（1~255）。recv_fd读s管道中所有字节直至null字符。null字符之前的所有字符都传送给调用者的*userfunc*。recv_fd读到的下一个字节是status字节。若status字节为0，那么一个描述符已传送过来，否则表示没有描述符可接收。

send_err函数在将出错消息写到STREAMS管道后，即调用send_fd函数。如程序清单17-11所示。

程序清单17-11　send_err函数

603

```
#include "apue.h"

/*
 * Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead.  We send the error back
 * using the send_fd()/recv_fd() protocol.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int     n;

    if ((n = strlen(msg)) > 0)
        if (writen(fd, msg, n) != n)    /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;   /* must be negative */

    if (send_fd(fd, errcode) < 0)
        return(-1);

    return(0);
}
```

以下两节介绍函数send_fd和recv_fd的具体实现。

17.4.1　经由基于STREAMS的管道传送文件描述符

文件描述符用两个ioctl命令经由STREAMS管道交换，这两个命令是：I_SENDFD和I_RECVFD。为了发送一个描述符，将ioctl的第三个参数设置为实际描述符。这示于程序清单17-12中。

程序清单17-12　STREAMS管道的send_fd函数

```
#include "apue.h"
#include <stropts.h>

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    char    buf[2];     /* send_fd()/recv_fd() 2-byte protocol */

    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (fd_to_send < 0) {
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
```

```
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        buf[1] = 0;     /* zero status means OK */
    }

    if (write(fd, buf, 2) != 2)
        return(-1);

    if (fd_to_send >= 0)
        if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
    return(0);
}
```

604

当接收一个描述符时，ioctl的第三个参数是一指向strrecvfd 结构的指针。

```
struct strrecvfd {
    int     fd;       /* new descriptor */
    uid_t   uid;      /* effective user ID of sender */
    gid_t   gid;      /* effective group ID of sender */
    char    fill[8];
};
```

recv_fd读STREAMS管道直到接收到双字节协议的第一个字节（null字节）。当发出I_RECVFD ioctl命令时，位于流首读队列中的下一条消息应当是一个描述符，它是由I_SENDFD发来的，或者是一条出错消息。该函数示于程序清单17-13中。

程序清单17-13 STREAMS管道的recv_fd函数

```
#include "apue.h"
#include <stropts.h>

/*
 * Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
 * 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int                 newfd, nread, flag, status;
    char                *ptr;
    char                buf[MAXLINE];
    struct strbuf       dat;
    struct strrecvfd    recvfd;

    status = -1;
    for ( ; ; ) {
        dat.buf = buf;
        dat.maxlen = MAXLINE;
        flag = 0;
        if (getmsg(fd, NULL, &dat, &flag) < 0)
            err_sys("getmsg error");
        nread = dat.len;
        if (nread == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
```

605

```
         * See if this is the final data with null & status.
         * Null must be next to last byte of buffer, status
         * byte is last byte.  Zero status means there must
         * be a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (ioctl(fd, I_RECVFD, &recvfd) < 0)
                        return(-1);
                    newfd = recvfd.fd;  /* new descriptor */
                } else {
                    newfd = -status;
                }
                nread -= 2;
            }
        }
        if (nread > 0)
            if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                return(-1);

        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

17.4.2 经由UNIX域套接字传送文件描述符

为了用UNIX域套接字交换文件描述符，调用sendmsg(2)和recvmsg(2)函数（16.5节）。这两个函数的参数中都有一个指向msghdr结构的指针，该结构包含了所有有关收发内容的信息。该结构的定义大致如下：

```
struct msghdr {
    void         *msg_name;        /* optional address */
    socklen_t     msg_namelen;     /* address size in bytes */
    struct iovec *msg_iov;         /* array of I/O buffers */
    int           msg_iovlen;      /* number of elements in array */
    void         *msg_control;     /* ancillary data */
    socklen_t     msg_controllen;  /* number of ancillary bytes */
    int           msg_flags;       /* flags for received message */
};
```

606

其中，头两个元素通常用于在网络连接上发送数据报文，在这里，目的地址可以由每个数据报文指定。下面两个元素使我们可以指定由多个缓冲区构成的数组（散布读和聚集写），这与对readv和writev函数（见14.7节）的说明一样。msg_flags字段包含了说明所接收到消息的标志，这些标志摘要示于表16-9中。

两个元素处理控制信息的传送和接收。msg_control字段指向cmsghdr（控制信息首部）结构，msg_controllen字段包含控制信息的字节数。

```
struct cmsghdr {
    socklen_t  cmsg_len;    /* data byte count, including header */
    int        cmsg_level;  /* originating protocol */
    int        cmsg_type;   /* protocol-specific type */
```

```
    /* followed by the actual control message data */
};
```

为了发送文件描述符，将cmsg_len设置为cmsghdr结构的长度加一个整型（描述符）的长度，cmsg_level字段设置为SOL_SOCKET，cmsg_type字段设置为SCM_RIGHTS，用以指明我们在传送访问权。（SCM指的是套接字级控制消息，socket_level control message。）访问权仅能通过UNIX域套接字传送。描述符紧随cmsg_type字段之后存放，用CMSG_DATA宏获得该整型量的指针。

三个宏用于访问控制数据，一个宏用于帮助计算cmsg_len所使用的值。

```
#include <sys/socket.h>

unsigned char *CMSG_DATA(struct cmsghdr *cp);
```

返回值：指向与cmsghdr结构相关联的数据的指针

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

返回值：指向与msghdr结构相关联的第一个cmsghdr结构的指针，若无这样的结构则返回NULL

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);
```

返回值：指向与msghdr结构相关联的下一个cmsghdr结构的指针，该msghdr结构给出了当前cmsghdr结构，若当前cmsghdr结构已是最后一个则返回NULL

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

返回值：为*nbytes*大小的数据对象分配的长度

Single UNIX规范定义了前三个宏，但没有定义CMSG_LEN。

CMSG_LEN宏返回为存放长度为*nbytes*的数据对象所需的字节数。它先将*nbytes*加上cmsghdr结构的长度，然后按处理机体系结构的对齐要求进行调整，最后再向上取整。 607

程序清单17-14是UNIX域套接字的send_fd函数。

程序清单17-14 UNIX域套接字的send_fd函数

```
#include "apue.h"
#include <sys/socket.h>

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
```

```
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send;     /* the fd to pass */
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

608

在sendmsg调用中，发送双字节协议数据（null和status字节）和描述符。

为了接收文件描述符（程序清单17-15），我们为cmsghdr结构和描述符分配足够大的空间，将msg_control指向该存储空间，然后调用recvmsg。我们使用CMSG_LEN宏计算所需空间的总量。

我们从UNIX域套接字读入，直至读到null字节，它位于最后的status字节之前。null字节之前是一条来自发送者的出错消息。这示于程序清单17-15中。

程序清单17-15　UNIX域套接字的recv_fd函数

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, seize_t (*userfunc)(int, const void *, size_t))
{
    int             newfd, nr, status;
    char            *ptr;
    char            buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;
```

```
    status = -1;
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_sys("recvmsg error");
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr);
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

609

注意，该程序总是准备接收一描述符（在每次调用recvmsg之前，设置msg_control和msg_controllen），但是仅当在返回时，msg_controllen非0，才确实接收到一描述符。

在传送文件描述符方面，UNIX域套接字和STREAMS管道之间的一个区别是，用STREAMS管道时我们得到发送进程的身份。某些UNIX域套接字版本提供类似的功能，但它们的接口不同。

> FreeBSD 5.2.1和Linux 2.4.22支持在UNIX域套接字上发送凭证，但实现方式不同。Mac OS 10.3是部分地从FreeBSD派生出来的，但禁止传送凭证。Solaris 9不支持在UNIX域套接字上传送凭证。

在FreeBSD，将凭证作为cmsgcred结构传送。

```
#define CMGROUP_MAX 16

struct cmsgcred {
    pid_t   cmcred_pid;                   /* sender's process ID */
    uid_t   cmcred_uid;                   /* sender's real UID */
    uid_t   cmcred_euid;                  /* sender's effective UID */
    gid_t   cmcred_gid;                   /* sender's real GID */
    short   cmcred_ngroups;               /* number of groups */
    gid_t   cmcred_groups[CMGROUP_MAX];   /* groups */
};
```

610

当传送凭证时，仅需为cmsgcred结构保留存储空间。内核将填充该结构以防止应用程序伪装成具有另一种身份。

在Linux中，将凭证作为ucred结构传送。

```
struct ucred {
    uint32_t  pid;  /* sender's process ID */
    uint32_t  uid;  /* sender's user ID */
    uint32_t  gid;  /* sender's group ID */
};
```

不同于FreeBSD的是，Linux要求在传送前先将结构初始化。内核将确保应用程序使用对应于调用程序的值，或具有适当的权限使用其他值。

程序清单17-16显示了更新后的send_fd函数，它包含了传送进程的凭证。

程序清单17-16　在UNIX域套接字上发送凭证

```
#include "apue.h"
#include <sys/socket.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
#define CREDSTRUCT      ucred
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct CREDSTRUCT   *credp;
    struct cmsghdr      *cmp;
    struct iovec        iov[1];
    struct msghdr       msg;
    char                buf[2]; /* send_fd/recv_ufd 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
```

```
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    msg.msg_flags = 0;
    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        cmp = cmptr;
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_RIGHTS;
        cmp->cmsg_len    = RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send;    /* the fd to pass */

        cmp = CMSG_NXTHDR(&msg, cmp);
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_CREDTYPE;
        cmp->cmsg_len    = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS)
        credp->uid = geteuid();
        credp->gid = getegid();
        credp->pid = getpid();
#endif
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

611

注意，只是在Linux上才需要初始化凭证结构。

程序清单17-17中的recv_ufd函数是recv_fd的修改版，它通过一引用参数返回发送者的用户ID。

程序清单17-17 在UNIX域套接字上接收凭证

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */
#include <sys/un.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define CR_UID          cmcred_uid
#define CREDOPT         LOCAL_PEERCRED
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
#define CREDSTRUCT      ucred
#define CR_UID          uid
#define CREDOPT         SO_PASSCRED
#define SCM_CREDTYPE    SCM_CREDENTIALS
```

612

```
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_ufd(int fd, uid_t *uidptr,
         ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr      *cmp;
    struct CREDSTRUCT   *credp;
    int                 newfd, nr, status;
    char                *ptr;
    char                buf[MAXLINE];
    struct iovec        iov[1];
    struct msghdr       msg;
    const int           on = 1;

    status = -1;
    newfd = -1;
    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt failed");
        return(-1);
    }
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_sys("recvmsg error");
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
```

613

```
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");

                    /* process the control data */
                    for (cmp = CMSG_FIRSTHDR(&msg);
                      cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                        if (cmp->cmsg_level != SOL_SOCKET)
                            continue;
                        switch (cmp->cmsg_type) {
                        case SCM_RIGHTS:
                            newfd = *(int *)CMSG_DATA(cmp);
                            break;
                        case SCM_CREDTYPE:
                            credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                            *uidptr = credp->CR_UID;
                        }
                    }
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

在FreeBSD上，我们指定SCM_CREDS来传送凭证；在Linux上，则使用SCM_CREDENTIALS。 614

17.5 open服务器版本1

使用文件描述符传送技术，我们开发了一个open服务器：一个由一个进程执行以打开一个或几个文件的程序。该服务器不是将文件内容送回调用进程，而是送回一个打开文件描述符。这使该服务器对任何类型的文件（例如一个设备或套接字）而不单是普通文件都能起作用。这也意味着，用IPC交换了最小量的信息——从客户进程到服务器进程传送文件名和打开模式，而从服务器进程到客户进程返回描述符。文件内容不需用IPC传送。

将服务器设计成一个单独的可执行程序（或者是由客户进程执行的，如本节所述；或者是一个守护服务器，我们将在下一节开发），有很多优点：

- 任一客户进程都易于和服务器进程联系，这类似于客户进程调用一库函数。不需要将一特定服务硬编码到应用程序中，而是设计一种可供重用的设施。
- 如若需要更改服务器，那么也只影响一个程序。相反，更新一库函数可能要更改调用此库函数的所有程序（用连编程序重新连接）。共享库函数可以简化这种更新（7.7节）。
- 服务器可以是设置用户ID程序，于是使其具有客户进程没有的附加权限。注意，一个库函数（或共享库函数）不能提供这种能力。

客户进程创建一个s管道（或者是一个基于STREAMS的管道，或者是UNIX域套接字对），然后调用fork和exec以调用服务器进程。客户进程经s管道发送请求，服务器进程经s管道回送响应。

定义客户进程和服务器进程间的应用程序协议如下：

(1) 客户进程通过s管道向服务器进程发送“open *<pathname>* *<openmode>*\0”形式的请求，*<openmode>*是open函数的第二个参数，以ASCII十进制数表示。该请求字符串以null字节结尾。

(2) 服务器进程调用send_fd 或send_err回送一打开描述符或一条出错消息。

下面是一个进程向其父进程发送一打开描述符的实例。17.6节将修改此实例，使其使用一个守护服务器进程，它将一个描述符发送给一个完全无关的进程。

程序清单17-18是头文件open.h，它包括标准头文件，并且定义了各个函数原型。

程序清单17-18　open.h头文件

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open"          /* client's request for server */

int     csopen(char *, int);
```

615

程序清单17-19是main函数，其中包含一个循环，它先从标准输入读一个路径名，然后将该文件复制至标准输出。它调用csopen以与open服务器进程联系，从其返回一个打开描述符。

程序清单17-19　客户进程main函数版本1

```
#include    "open.h"
#include    <fcntl.h>

#define BUFFSIZE    8192

int
main(int argc, char *argv[])
{
    int     n, fd;
    char    buf[BUFFSIZE], line[MAXLINE];

    /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* replace newline with null */

        /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue;   /* csopen() prints error from server */

        /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }

    exit(0);
}
```

程序清单17-20是函数csopen，它先创建一个s管道，然后进行服务器进程的fork和exec操作。

程序清单17-20 csopen函数版本1

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      fd[2] = { -1, -1 };

    if (fd[0] < 0) {    /* fork/exec our open server first time */
        if (s_pipe(fd) < 0)
            err_sys("s_pipe error");
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO &&
              dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            if (fd[1] != STDOUT_FILENO &&
              dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            if (execl("./opend", "opend", (char *)0) < 0)
                err_sys("execl error");
        }
        close(fd[1]);               /* parent */
    }
    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";      /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len)
        err_sys("writev error");

    /* read descriptor, returned errors handled by write() */
    return(recv_fd(fd[0], write));
}
```

616

子进程关闭管道的一端，父进程关闭另一端。作为服务器进程，子进程也将管道的一端复制到其标准输入和标准输出。(另一种备选方案是将描述符fd[1]的ASCII表示形式作为一个参数传送给服务器进程。)

父进程将请求发送给服务器进程，请求中包含路径名和打开模式。最后，父进程调用recv_fd返回描述符或错误消息。如果服务器进程返回一错误消息，那么，父进程调用write，向标准出错输出该消息。

现在，观察open服务器进程。其程序是opend，它由子进程执行（见程序清单17-20)。先

观察opend.h头文件（见程序清单17-21），它包括了标准头文件，并且声明了全局变量和函数原型。

617

程序清单17-21 opend.h头文件版本1

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open"          /* client's request for server */

extern char  errmsg[];   /* error message string to return to client */
extern int   oflag;      /* open() flag: O_xxx ... */
extern char *pathname;   /* of file to open() for client */

int     cli_args(int, char **);
void    request(char *, int, int);
```

main函数（程序清单17-22）经s管道（它的标准输入）读来自客户进程的请求，然后调用函数request。

程序清单17-22 服务器进程main函数版本1

```
#include    "opend.h"

char    errmsg[MAXLINE];
int     oflag;
char   *pathname;

int
main(void)
{
    int     nread;
    char    buf[MAXLINE];

    for ( ; ; ) {   /* read arg buffer from client, process request */
        if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break;      /* client has closed the stream pipe */
        request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

程序清单17-23中的request函数承担了全部工作。它调用函数buf_args将客户进程请求分解成标准argv型的参数表，然后调用函数cli_args处理客户进程的参数。如果一切正常，则调用open打开相应文件，接着调用send_fd，经由s管道（它的标准输出）将描述符回送给客户进程。如果出错则调用send_err回送一则出错消息，其中使用了前面说明的客户进程-服务器进程协议。

程序清单17-23 request函数版本1

```
#include    "opend.h"
#include    <fcntl.h>

void
request(char *buf, int nread, int fd)
{
  int     newfd;
```

618

```
    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request not null terminated: %*.*s\n",
          nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }
    if (buf_args(buf, cli_args) < 0) {  /* parse args & set options */
        send_err(fd, -1, errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n", pathname,
          strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }
    if (send_fd(fd, newfd) < 0)     /* send the descriptor */
        err_sys("send_fd error");
    close(newfd);       /* we're done with descriptor */
}
```

客户进程请求是一个以null结尾的字符串，它所包含的各参数由空格分隔。程序清单17-24中的buf_args函数将字符串分解成标准argv型参数表，并调用用户函数处理参数。本节稍后将用到该函数。我们使用ISO C函数strtok将字符串分割成参数。

程序清单17-24 buf_args函数

```
#include "apue.h"

#define MAXARGC     50  /* max number of arguments in buf */
#define WHITE   " \t\n" /* white space for tokenizing arguments */

/*
 * buf[] contains white-space-separated arguments.  We convert it to an
 * argv-style array of pointers, and call the user's function (optfunc)
 * to process the array.  We return -1 if there's a problem parsing buf,
 * else we return whatever optfunc() returns.  Note that user's buf[]
 * array is modified (nulls placed after each token).
 */
int
buf_args(char *buf, int (*optfunc)(int, char **))
{
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL)     /* an argv[0] is required */
        return(-1);
    argv[argc = 0] = buf;
    while ((ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)    /* -1 for room for NULL at end */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    /*
     * Since argv[] pointers point into the user's buf[],
     * user's function can just copy the pointers, even
     * though argv[] array will disappear on return.
     */
```

619

```
    return((*optfunc)(argc, argv));
}
```

buf_args调用的服务器函数是cli_args（见程序清单17-25）。它验证客户进程发送的参数个数是否正确，然后将路径名和打开模式存放在全局变量中。

程序清单17-25　cli_args函数

```
#include    "opend.h"

/*
 * This function is called by buf_args(), which is called by
 * request().  buf_args() has broken up the client's buffer
 * into an argv[]-style array, which we now process.
 */
int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }
    pathname = argv[1];     /* save ptr to pathname to open */
    oflag = atoi(argv[2]);
    return(0);
}
```

这样由客户进程执行fork和exec而调用的open服务器就完成了。在fork之前创建了一个s管道，然后客户进程和服务器进程用它进行通信。在这种安排下，每个客户进程都有一服务器进程。

17.6　open服务器版本2

620　在上一节中，我们开发了一个open服务器，客户进程调用fork和exec来调用该服务器，它演示了如何从子进程向父进程传送文件描述符。本节将开发一个以守护进程方式运行的open服务器。用一个服务器进程处理所有客户进程的请求。这一设计应该更加有效，因为没有使用fork和exec。在客户进程和服务器进程之间仍使用s管道，来演示在无关进程之间如何传送文件描述符。我们仍将使用17.2.2节说明的三个函数：serv_listen、serv_accept和cli_conn。这一服务器还将展示了一个服务器进程如何向多个客户进程提供服务，其中使用了14.5节中介绍过的select和poll函数。

本节所述的客户进程类似于17.5节。确实，文件main.c是完全相同的（见程序清单17-19）。在open.h头文件（见程序清单17-18）中则加了下面一行：

```
#define CS_OPEN "/home/sar/opend"   /* server's well-known name */
```

因为在这里调用的是cli_conn而非fork和exec，所以文件open.c与程序清单17-20中的不同。这示于程序清单17-26中。

程序清单17-26　csopen函数第2版

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
```

```
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      csfd = -1;

    if (csfd < 0) {     /* open connection to conn server */
        if ((csfd = cli_conn(CS_OPEN)) < 0)
            err_sys("cli_conn error");
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";  /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* null always sent */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(csfd, &iov[0], 3) != len)
        err_sys("writev error");

    /* read back descriptor; returned errors handled by write() */
    return(recv_fd(csfd, write));
}
```

621

从客户进程到服务器进程所使用的协议仍然相同。

我们再来看服务器进程。头文件opend.h（见程序清单17-27）包括了标准头文件，并且声明了全局变量和函数原型。

程序清单17-27 open.h头文件版本2

```
#include "apue.h"
#include <errno.h>

#define CS_OPEN "/home/sar/opend"   /* well-known name */
#define CL_OPEN "open"              /* client's request for server */

extern int   debug;      /* nonzero if interactive (not daemon) */
extern char  errmsg[];   /* error message string to return to client */
extern int   oflag;      /* open flag: O_xxx ... */
extern char *pathname;   /* of file to open for client */

typedef struct {     /* one Client struct per connected client */
  int   fd;          /* fd, or -1 if available */
  uid_t uid;
} Client;

extern Client   *client;        /* ptr to malloc'ed array */
extern int       client_size;   /* # entries in client[] array */

int      cli_args(int, char **);
int      client_add(int, uid_t);
void     client_del(int);
void     loop(void);
void     request(char *, int, int, uid_t);
```

因为此服务器进程处理所有客户进程，所以它必须保存每个客户进程连接的状态。这是用opend.h头文件中声明的client数组实现的。程序清单17-28定义了三个操纵此数组的函数。

程序清单17-28 操纵client数组的三个函数

```
#include    "opend.h"

#define NALLOC  10      /* # client structs to alloc/realloc for */

static void
client_alloc(void)      /* alloc more entries in the client[] array */
{
    int     i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size+NALLOC)*sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

    /* initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;  /* fd of -1 means entry available */

    client_size += NALLOC;
}

/*
 * Called by loop() when connection request from a new client arrives.
 */
int
client_add(int fd, uid_t uid)
{
    int     i;

    if (client == NULL)     /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) {   /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i);  /* return index in client[] array */
        }
    }

    /* client array full, time to realloc for more */
    client_alloc();
    goto again;     /* and search again (will work this time) */
}

/*
 * Called by loop() when we're done with a client.
 */
void
client_del(int fd)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
```

622

```
            }
        }
        log_quit("can't find client entry for fd %d", fd);
    }
```

第一次调用client_add时，它调用client_alloc，client_alloc又调用malloc为该数组的10个登记项分配空间。在这10个登记项全部用完后，如若再调用client_add，那么在client_alloc函数中将调用realloc，由该函数分配附加空间。依靠这种动态空间分配，我们无需将编译时client数组的长度限定为某个估计值，并将该值写入头文件。如果出错，那么因为假定服务器进程是守护进程，所以这些函数调用log_函数（见附录B）。

623

main函数（见程序清单17-29）定义全局变量，处理命令行选项，然后调用loop函数。如果以-d选项调用服务器进程，则它以交互方式运行而不是守护进程。当测试些服务器进程时，使用交互运行方式。

程序清单17-29 服务器进程main函数版本2

```
#include    "opend.h"
#include    <syslog.h>

int      debug, oflag, client_size, log_to_stderr;
char     errmsg[MAXLINE];
char    *pathname;
Client  *client = NULL;

int
main(int argc, char *argv[])
{
    int     c;

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ((c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = log_to_stderr = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemonize("opend");

    loop();     /* never returns */
}
```

loop函数是服务器进程的无限循环。我们将给出该函数的两种版本。程序清单17-30是使用select的一种版本；程序清单17-31是使用poll的另一种版本。

程序清单17-30 使用select的loop函数

```
#include    "opend.h"
#include    <sys/time.h>
#include    <sys/select.h>
```

```
void
loop(void)
{
    int     i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;

    FD_ZERO(&allset);

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    FD_SET(listenfd, &allset);
    maxfd = listenfd;
    maxi = -1;

    for ( ; ; ) {
        rset = allset;  /* rset gets modified each time around */
        if ((n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
            log_sys("select error");

        if (FD_ISSET(listenfd, &rset)) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            FD_SET(clifd, &allset);
            if (clifd > maxfd)
                maxfd = clifd;  /* max fd for select() */
            if (i > maxi)
                maxi = i;   /* max index in client[] array */
            log_msg("new connection: uid %d, fd %d", uid, clifd);
            continue;
        }

        for (i = 0; i <= maxi; i++) {   /* go through client[] array */
            if ((clifd = client[i].fd) < 0)
                continue;
            if (FD_ISSET(clifd, &rset)) {
                /* read argument buffer from client */
                if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", clifd);
                } else if (nread == 0) {
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, clifd);
                    client_del(clifd);  /* client has closed cxn */
                    FD_CLR(clifd, &allset);
                    close(clifd);
                } else {    /* process client's request */
                    request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

624

625

此函数调用serv_listen创建服务器进程对于客户进程连接的端点。此函数的其余部分是一个循环，它从select调用开始。在select返回后，可能发生下列两种情况：

(1) 描述符listenfd可能准备好读，这意味着新客户进程已调用了cli_conn。为了处理这

种情况，我们将调用serv_accept，然后为新客户进程更新client数组以及相关的簿记消息。（跟踪记录作为select第一个参数的最高描述符编号，也记下client数组中用到的最大下标。）

(2) 一个现存的客户进程的连接可能准备好读。这意味着该客户进程已经终止，或者已经发送了一个新请求。如果read返回0（文件结束），则可认为一客户进程终止。如果read返回值大于0则可判定有一新请求等待处理，调用request处理此新的客户进程请求。

用allset描述符集跟踪当前使用的描述符。当新客户进程连至服务器进程时，此描述符集的某个适当位被打开。当该客户进程终止时，这个位就被关闭。

因为客户进程的所有描述符（包括与服务器进程的连接）都由内核自动关闭，所以我们总能知道什么时候一客户进程终止，该终止是否自愿。这与XSI IPC机制不同。

使用poll的loop函数示于程序清单17-31中。

程序清单17-31 使用poll的loop函数

```
#include    "opend.h"
#include    <poll.h>
#if !defined(BSD) && !defined(MACOS)
#include    <stropts.h>
#endif

void
loop(void)
{
    int             i, maxi, listenfd, clifd, nread;
    char            buf[MAXLINE];
    uid_t           uid;
    struct pollfd   *pollfd;

    if ((pollfd = malloc(open_max() * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0);    /* we use [0] for listenfd */
    pollfd[0].fd = listenfd;
    pollfd[0].events = POLLIN;
    maxi = 0;

    for ( ; ; ) {
        if (poll(pollfd, maxi + 1, -1) < 0)
            log_sys("poll error");

        if (pollfd[0].revents & POLLIN) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            pollfd[i].fd = clifd;
            pollfd[i].events = POLLIN;
            if (i > maxi)
                maxi = i;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i <= maxi; i++) {
            if ((clifd = client[i].fd) < 0)
                continue;
            if (pollfd[i].revents & POLLHUP) {
```

626

```
                goto hungup;
            } else if (pollfd[i].revents & POLLIN) {
                /* read argument buffer from client */
                if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", clifd);
                } else if (nread == 0) {
hungup:
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, clifd);
                    client_del(clifd);  /* client has closed conn */
                    pollfd[i].fd = -1;
                    close(clifd);
                } else {        /* process client's request */
                    request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

为使客户进程数量能与打开描述符的数量相当，我们动态地为pollfd结构的数组分配空间(参见程序清单2-4中的函数open_max)。

client数组中下标号为0的登记项用于listenfd描述符。于是，client数组中的客户进程下标号与pollfd数组中所用的下标号相同。新客户进程连接的到达由listenfd描述符中的POLLIN指示。如同前述，调用serv_accept以接收该连接。

对于一个现存的客户进程，我们应当处理来自poll的两个不同事件：由POLLHUP指示的客户进程终止；由POLLIN指示的来自现存客户进程的一个新请求。回忆习题15.7，在流还有数据可读时，挂起消息可能到达流首。对于管道，我们希望在处理挂起前先读所有数据。但是对于服务器进程，当从客户进程接收到挂起消息时，它可以用close关闭至客户进程的连接(流)，于是也就丢弃了仍在流上的所有数据。因为已经不能回送任何响应，所以也就没有理由再去处理仍在流上的任何请求。 627

如同此函数的select版本，我们调用request函数处理来自客户进程的新请求（程序清单17-32)。此函数类似于其早期版本（程序清单17-23)。它调用同一函数buf_args（程序清单17-24)，后者又调用cli_args（程序清单17-25)，但是因为它是在一个守护进程中运行的，所以出错消息是在日志文件中记录，而不是在标准出错流上打印。

程序清单17-32　request函数版本2

```
#include    "opend.h"
#include    <fcntl.h>

void
request(char *buf, int nread, int clifd, uid_t uid)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg,
          "request from uid %d not null terminated: %*.*s\n",
          uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
```

```
    log_msg("request: %s, from uid %d", buf, uid);

    /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ((newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
          pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* send the descriptor */
    if (send_fd(clifd, newfd) < 0)
        log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd);       /* we're done with descriptor */
}
```

这样就完成了open服务器进程版本2，它使用单个守护进程处理所有的客户进程请求。 628

17.7 小结

本章的重点是如何实现在进程间传送文件描述符，以及怎样使服务器进程接受来自众多客户进程的唯一连接。本章说明了如何使用基于STREAMS的管道以及UNIX域套接字实现这些功能。虽然所有平台都支持UNIX域套接字（参见表15-1），但是各种实现都有不同之处，这使我们很难开发可移植的应用程序。

本章给出了open服务器进程的两个版本。一个版本由客户进程用fork和exec直接调用，另一版本是处理所有客户进程请求的守护服务器进程。这两个版本均采用文件描述符传送和接收函数。后一版本还采用了17.2.2节所述的客户进程–服务器进程连接函数以及14.5节所述的I/O多路转接函数。

习题

17.1 改写程序清单17-1，对于STREAMS管道使用标准I/O库函数代替read和write。

17.2 使用本章说明的文件描述符传送函数以及8.9节中说明的父–子进程同步例程，编写具有下列功能的程序：该程序调用fork，然后子进程用open打开一现存文件并将打开文件描述符传送给父进程。接着，子进程调用lseek确定该文件当前读、写位置，并通知父进程。父进程读该文件的当前偏移量，并打印它以便验证。若此文件如上所述从子进程传递到父进程，则父、子进程应共享同一文件表项，所以当子进程每次更改该文件当前偏移量，都会同样影响到父进程的描述符。使子进程将该文件定位至一个不同的偏移量，并再次通知父进程。

17.3 程序清单17-21和17-22分别定义和声明了全局变量，两者的区别是什么？

17.4 改写buf_args函数（见程序清单17-24），删除其中对argv数组长度的编译时间限制。请用动态存储分配。

17.5 说明优化程序清单17-30和17-31中loop函数的方法，并实现之。 629

第18章

终端I/O

18.1 引言

所有操作系统的终端I/O处理都是非常繁琐的，UNIX也不例外。在大多数版本的UNIX手册中，终端I/O手册页常常是最长的部分。

20世纪70年代后期，UNIX系统Ⅲ开发了与传统的V7不同的一组终端例程，从而形成了UNIX终端I/O处理的两种不同风格，系统Ⅲ的风格一直延续至系统V；V7的风格则成为BSD类系统的标准风格。POSIX.1在这两种风格的基础上制定了终端I/O标准。本章将介绍POSIX.1的终端函数，以及某些平台特有的增加部分。

终端I/O的用途很广泛，包括用于终端、计算机之间的直接连线、调制解调器以及打印机等等，所以终端I/O系统非常复杂。

18.2 综述

终端I/O有两种不同的工作模式：

(1) 规范模式输入处理（Canonical mode input processing）。在这种模式中，终端输入以行为单位进行处理。对于每个读要求，终端驱动程序最多返回一行。

(2) 非规范模式输入处理（Noncanonical mode input processing）。输入字符并不组成行。

631

如果不作特殊处理，则默认模式是规范模式。例如，若shell把标准输入重定向到终端，在用`read`和`write`将标准输入复制到标准输出时，终端以规范模式进行工作，每次`read`最多返回一行。操纵整个屏幕的程序（例如`vi`编辑程序）使用非规范模式，原因是它的命令是由一个或几个字符组成的，并且不以换行符终止。另外，该编辑程序使用了若干特殊字符作为编辑命令，所以它也不希望系统对特殊字符进行处理。例如，Ctrl+D字符通常是终端的文件结束符，但在`vi`中它是向下滚动半个屏幕的命令。

> V7和较早BSD风格类的终端驱动程序支持三种终端输入模式：(a) 精细加工模式（输入组成行，并对特殊字符进行处理）；(b) 原始模式（输入不组成行，也不对特殊字符进行处理）；(c) cbreak模式（输入不组成行，但对某些特殊字符进行处理）。程序清单18-10显示了将终端设置为cbreak或原始模式的POSIX.1函数。

POSIX.1定义了11个特殊输入字符，其中9个可以改变。本书已经用到了其中几个，例如文件结束符（通常是Ctrl+D）、挂起字符（通常是Ctrl+Z）。18.3节将对其中每个字符进行说明。

终端设备是由一般位于内核中的终端驱动程序控制的。每个终端设备有一个输入队列和一个输出队列，如图18-1所示。

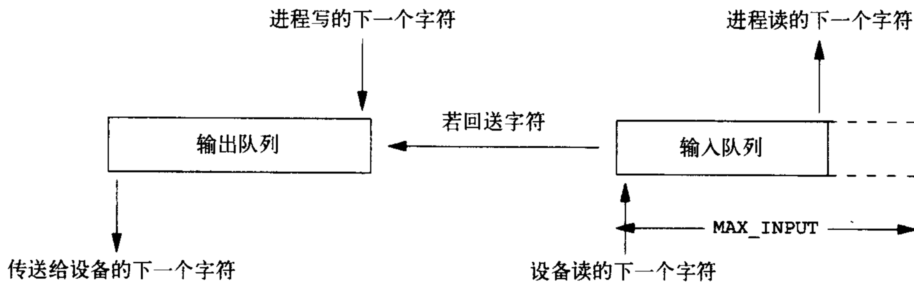


图18-1　终端设备的输入、输出队列逻辑结构

对此图要说明下列几点：

- 如果打开了回显功能，则在输入队列和输出队列之间有一个隐含的连接。
- 输入队列的长度MAX_INPUT（见表2-11）是有限值，当一个特定设备的输入队列已经填满时，系统对此作何种处理依赖于实现。大多数UNIX系统的处理方式是回显响铃字符。

632

- 图中没有显示另一个输入限制MAX_CANON，它是在一个规范输入行中的最大字节数。
- 虽然输出队列通常也是有限长度，但是程序并不能获得这个定义其长度的常量，这是因为当输出队列将要填满时，内核使写进程休眠直至写队列中有可用的空间，所以程序无需关心该队列的长度。
- 我们将说明如何使用tcflush函数刷清（flush）输入或输出队列。与此类似，在说明tcsetattr函数时，我们将会了解到如何通知系统只有在输出队列为空时才改变一个终端设备的属性。（例如，想要改变输出属性时就要这样做。）我们也能通知系统，当它正在改变终端属性时，要丢弃在输入队列中的一切东西。（如果正在改变输入属性，或者在规范和非规范模式之间进行转换，则可能希望这样做，以免以错误的模式对以前输入的字符进行解释。）

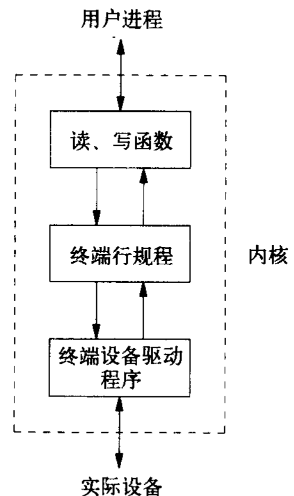


图18-2　终端行规程

大多数UNIX系统在一个称为*终端行规程*（terminal line discipline）的模块中进行规范处理。它是位于内核通用读、写函数和实际设备驱动程序之间的模块（见图18-2）。

633

注意，此图与关于流模块的图14-5非常类似。在第19章讨论伪终端时还将使用此图。

所有我们可以检测和更改的终端设备特性都包含在termios结构中。该结构定义在头文件<termios.h>中，本章经常使用这一头文件。

```
struct termios {
  tcflag_t  c_iflag;     /* input flags */
  tcflag_t  c_oflag;     /* output flags */
  tcflag_t  c_cflag;     /* control flags */
  tcflag_t  c_lflag;     /* local flags */
  cc_t      c_cc[NCCS];  /* control characters */
};
```

粗略而言，输入标志由终端设备驱动程序用来控制字符的输入（剥除输入字节的第8位，允许输入奇偶校验等等），输出标志则控制驱动程序输出（执行输出处理、将换行符映射为CR/LF等），

控制标志影响到RS-232串行线（忽略调制解调器的状态线、每个字符的一个或两个停止位等等），本地标志影响驱动程序和用户之间的接口（回送的开或关、可视的擦除字符、终端产生的信号的启用以及对后台输出的作业控制停止信号等）。

类型`tcflag_t`的长度足以保存每个标志值。它经常被定义为`unsigned int`或者`unsigned long`。`c_cc`数组包含了所有可以更改的特殊字符。`NCCS`是该数组的长度，一般介于15到20之间（大多数UNIX系统支持的特殊字符较POSIX所定义的11个要多）。`cc_t`类型的长度足以保存每个特殊字符，而且它往往是`unsigned char`型的。

> POSIX标准之前的系统V版本有一个名为`<termio.h>`的头文件、一个名为`termio`的数据结构。为了区别于这些老名字，POSIX.1在新名字后加了一个s。

表18-1至表18-4列出了所有可以进行更改以影响终端设备特性的终端标志。注意，虽然Single UNIX Specification定义了所有平台都支持的公共子集，但是各平台还有自己的扩充部分。这些扩充部分与系统各自不同的历史发展过程有关。18.5节将详细讨论这些标志值。

表18-1　`c_cflag`终端标志

标　志	说　明	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
CBAUDEXT	扩充的波特率					•
CCAR_OFLOW	输出的DCD流控制		•		•	
CCTS_OFLOW	输出的CTS流控制		•		•	•
CDSR_OFLOW	输出的DSR流控制		•		•	
CDTR_IFLOW	输出的DTR流控制		•		•	
CIBAUDEXT	扩充输入波特率					•
CIGNORE	忽略控制标志		•		•	
CLOCAL	忽略调制解调器状态行	•	•	•	•	•
CREAD	启用接收装置	•	•	•	•	•
CRTSCTS	启用硬件流控制		•	•	•	•
CRTS_IFLOW	输入的RTS流控制		•		•	•
CRTSXOFF	启用输入硬件流控制					•
CSIZE	字符大小屏蔽	•	•	•	•	•
CSTOPB	送两个停止位，否则为1位	•	•	•	•	•
HUPCL	最后关闭时断开	•	•	•	•	•
MDMBUF	与CCAR_OFLOW相同		•		•	
PARENB	进行奇偶校验	•	•	•	•	•
PAREXT	标记或空奇偶性					•
PARODD	奇校验，否则为偶校验	•	•	•	•	•

表18-2　`c_iflag`终端标志

标　志	说　明	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
BRKINT	接到BREAK时产生SIGINT	•	•	•	•	•
ICRNL	将输入的CR转换为NL	•	•	•	•	•
IGNBRK	忽略BREAK条件	•	•	•	•	•
IGNCR	忽略CR	•	•	•	•	•
IGNPAR	忽略奇偶错字符	•	•	•	•	•

（续）

标 志	说 明	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
IMAXBEL	在输入队列满时振铃		•	•	•	•
INLCR	将输入的NL转换为CR	•	•	•	•	•
INPCK	打开输入奇偶校验	•	•	•	•	•
ISTRIP	剥除输入字符的第8位	•	•	•	•	•
IUCLC	将输入的大写字符转换成小写字符			•		•
IXANY	使任一字符都重新启动输出	XSI	•	•	•	•
IXOFF	使启动/停止输入控制流起作用	•	•	•	•	•
IXON	使启动/停止输出控制流起作用	•	•	•	•	•
PARMRK	标记奇偶错	•	•	•	•	•

表18-3 c_lflag终端标志

标 志	说 明	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
ALTWERASE	使用替换WERASE算法		•		•	
ECHO	进行回送	•	•	•	•	•
ECHOCTL	回送控制字符为^(Char)		•	•	•	•
ECHOE	可见擦除符	•	•	•	•	•
ECHOK	回送kill符	•	•	•	•	•
ECHOKE	kill的可见擦除		•	•	•	•
ECHONL	回送NL	•	•	•	•	•
ECHOPRT	硬拷贝的可见擦除方式		•	•	•	•
EXTPROC	外部字符处理		•		•	
FLUSHO	刷清输出		•	•	•	•
ICANON	规范输入	•	•	•	•	•
IEXTEN	启用扩充的输入字符处理	•	•	•	•	•
ISIG	启用终端产生的信号	•	•	•	•	•
NOFLSH	在中断或退出键后禁用刷清	•	•	•	•	•
NOKERNINFO	由STATUS无内核输出		•		•	
PENDIN	重新打印未决输入		•	•	•	•
TOSTOP	对于后台输出发送SIGTTOU	•	•	•	•	•
XCASE	规范的大/小写表示			•		•

表18-4 c_oflag终端标志

标 志	说 明	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
BSDLY	退格延迟屏蔽	XSI		•		•
CMSPAR	标记或空奇偶性			•		
CRDLY	CR延迟屏蔽	XSI		•		•
FFDLY	换页延迟屏蔽	XSI		•		•
NLDLY	NL延迟屏蔽	XSI		•		•
OCRNL	将输出的CR转换为NL	XSI	•	•		•
OFDEL	填充符为DEL，否则为NUL	XSI		•		•
OFILL	对于延迟使用填充符	XSI		•		•

（续）

标　志	说　明	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
OLCUC	将输出的小写字符转换为大写字符			•		•
ONLCR	将NL转换为CR-NL	XSI	•	•	•	•
ONLRET	NL执行CR功能	XSI	•	•		•
ONOCR	在0列不输出CR	XSI	•	•		•
ONOEOT	在输出中删除EOT(^D)字符		•		•	
OPOST	执行输出处理	•	•	•	•	•
OXTABS	将制表符扩充为空格		•		•	
TABDLY	水平制表符延迟屏蔽	XSI		•		•
VTDLY	垂直制表符延迟屏蔽	XSI		•		•

给出了所有可用的选项后，如何才能检测和更改终端设备的这些特性呢？表18-5列出了Single UNIX Specification所定义的对终端设备进行操作的各个函数。（除tcgetsid是Single UNIX Specification的XSI扩展外，列出的其他函数都是POSIX规范的基本部分。9.7节已说明了tcgetpgrp、tcgetsid和tcsetpgrp函数。）

表18-5　终端I/O函数摘要

函　数	说　明
tcgetattr	取属性（termios结构）
tcsetattr	设置属性（termios结构）
cfgetispeed	得到输入速度
cfgetospeed	得到输出速度
cfsetispeed	设置输入速度
cfsetospeed	设置输出速度
tcdrain	等待所有输出都被传输
tcflow	挂起传输或接收
tcflush	刷清未决输入和/或输出
tcsendbreak	送BREAK字符
tcgetpgrp	得到前台进程组ID
tcsetpgrp	设置前台进程组ID
tcgetsid	得到控制TTY的会话首进程的进程组ID（XSI扩展）

注意，对终端设备，Single UNIX Specification没有使用经典的ioctl，而使用了表18-5中列出的13个函数。这样做的理由是：对于终端设备的ioctl函数，其最后一个参数的数据类型随执行动作的不同而不同。于是，这使得对参数进行类型检查成为不可能。

虽然对终端设备进行操作只有13个函数，但是表18-5中头两个函数（tcgetattr和tcsetattr）能处理大约70种不同的标志（见表18-1至表18-4）。对于终端设备有大量选项可供使用，此外，对于一个特定设备（终端、调制解调器、激光打印机等等）还要决定所需的选项，这些都使对终端设备的处理变得异常复杂。

表18-5中列出的13个函数之间的关系示于图18-3中。

POSIX.1没有规定在termios结构中何处存放波特率信息，那是具体实现的细节。某些系统（例如Linux和Solaris）将此信息存放在c_cflag字段中。BSD派生的系统（例如FreeBSD和Mac OS X）则在此结构中有两个分开的字段：一个存放输入速度，另一个则存放输出速度。

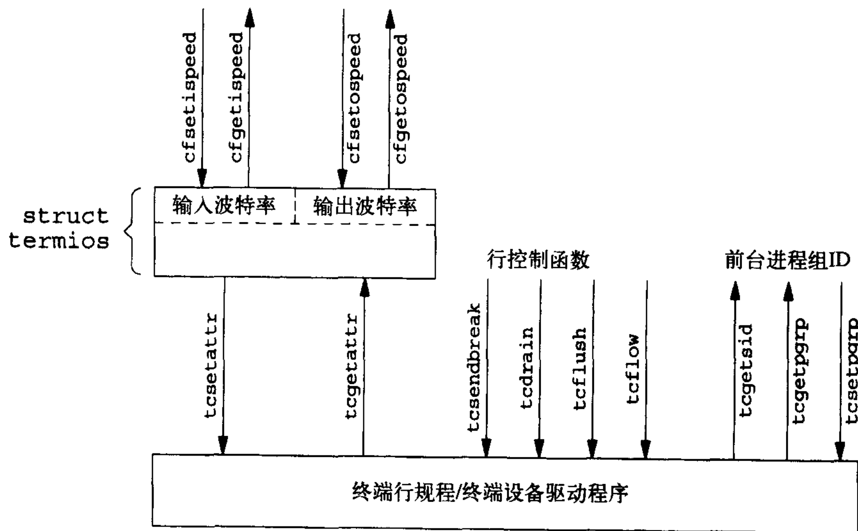


图18-3 与终端有关的函数之间的关系

634~637

18.3 特殊输入字符

POSIX.1定义了11个在输入时作特殊处理的字符。实现定义了另外一些特殊字符。表18-6摘要列出了这些特殊字符。

表18-6 终端特殊输入字符

字符	说明	下标	由……启用 字段	标志	典型值	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
CR	回车	(不能更改)	c_lflag	ICANON	\r	•	•	•	•	•
DISCARD	擦除输出	VDISCARD	c_lflag	IEXTEN	^O		•	•	•	•
DSUSP	延迟挂起 (SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		•		•	•
EOF	文件结束	VEOF	c_lflag	ICANON	^D	•	•	•	•	•
EOL	行结束	VEOL	c_lflag	ICANON		•	•	•	•	•
EOL2	供替换的行结束	VEOL2	c_lflag	ICANON			•	•	•	•
ERASE	向前擦除一个字符	VERASE	c_lflag	ICANON	^H, ^?	•	•	•	•	•
ERASE2	供替换的向前擦除字符	VERASE2	c_lflag	ICANON	^H, ^?		•			
INTR	中断信号 (SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•	•	•	•	•
KILL	擦行	VKILL	c_lflag	ICANON	^U	•	•	•	•	•
LNEXT	下一个字面字符	VLNEXT	c_lflag	IEXTEN	^V		•	•	•	•
NL	换行	(不能更改)	c_lflag	ICANON	\n	•	•	•	•	•
QUIT	退出信号 (SIGQUIT)	VQUIT	c_lflag	ISIG	^\	•	•	•	•	•
REPRINT	再打印全部输入	VREPRINT	c_lflag	ICANON	^R		•	•	•	•
START	恢复输出	VSTART	c_iflag	IXON/IXOFF	^Q	•	•	•	•	•
STATUS	状态请求	VSTATUS	c_lflag	ICANON	^T		•		•	
STOP	停止输出	VSTOP	c_iflag	IXON/IXOFF	^S	•	•	•	•	•
SUSP	挂起信号 (SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•	•	•	•	•
WERASE	擦除一个字	VWERASE	c_lflag	ICANON	^W		•	•	•	•

在POSIX.1的11个特殊字符中，可将其中9个更改为几乎任何值。不能更改的两个特殊字符

是换行符和回车符（\n和\r），有些实现也不允许更改STOP和START字符。为了进行修改，只要更改termios结构中c_cc数组的相应项。该数组中的元素都用名字作为下标进行引用，每个名字都以字母V开头（见表18-6中的第3列）。

POSIX.1允许禁用这些字符。若将c_cc数组中的某项设置为_POSIX_VDISABLE的值，则禁用相应的特殊字符。

Single UNIX Specification的早期版本中，支持_POSIX_VDISABLE是作为可选项的。现在则是作为要求项。

本书讨论的四种平台都支持此特性。Linux 2.4.22 和Solaris 9将_POSIX_VDISABLE定义为0，而FreeBSD 5.2.1和Mac OS X 10.3则将其定义为0xff。

某些早期的UNIX系统所用的方法是：若相应的特殊输入字符是0，则禁用该字符。

638

实 例

在详细说明各特殊字符之前，先看一个更改特殊字符的程序。程序清单18-1禁用中断字符，并将文件结束符设置为Ctrl+B。

程序清单18-1 禁用中断字符和更改文件结束字符

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios  term;
    long            vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("standard input is not a terminal device");

    if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
        err_sys("tcgetattr error");

    term.c_cc[VINTR] = vdisable;    /* disable INTR character */
    term.c_cc[VEOF]  = 2;           /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

对此程序要说明下列几点：

- 仅当标准输入是终端设备时才修改终端特殊字符。调用isatty（见18.9节）对此进行检测。
- 用fpathconf取_POSIX_VDISABLE值。
- 函数tcgetattr（见18.4节）从内核取termios结构。在修改了此结构后，调用tcsetattr函数设置属性，这样就可进行我们所希望的修改，而其他属性则保持不变。
- 禁用中断键与忽略中断信号是不同的。程序清单18-1所做的是禁止使用使终端驱动程序

639 产生SIGINT信号的特殊字符。但仍可使用kill函数向进程发送该信号。 □

下面较详细地说明各个特殊字符。我们称这些字符为特殊输入字符，但是其中STOP和START（Ctrl+S和Ctrl+Q）两个字符在输出时也对它们进行特殊处理。注意，这些字符中的大多数在被终端驱动程序识别并进行特殊处理后都被丢弃，并不将它们传送给执行读终端操作的进程。例外的字符是换行符（NL，EOL，EOL2）和回车符（CR）。

CR　回车符。不能更改此字符。以规范模式进行输入时识别此字符。当设置了ICANON（规范模式）和ICRNL（将CR映射为NL）以及没有设置IGNCR（忽略CR）时，将CR转换成NL，并产生与NL符相同的作用。
此字符返回给读进程（多半是在转换成NL后）。

DISCARD　删除符。在扩充模式下（IEXTEN），在输入中识别此字符。在输入另一个DISCARD符之前或删除条件被清除之前（见FLUSHO选项），此字符使后续输出都被删除。在处理后此字符即被删除，不送向读进程。

DSUSP　延迟-挂起作业控制字符（delayed-suspend job-control character）。在扩充方式下，若支持作业控制并且ISIG标志被设置，则在输入中识别此字符。与SUSP字符的相同处是：延迟-挂起字符产生SIGTSTP信号，它被送至前台进程组中的所有进程（参见图9-7）。但是延迟-挂起字符产生信号的时间并不是在键入此字符时，而是在一个进程读控制终端读到此字符时。在处理后，此字符即被删除，不送向读进程。

EOF　文件结束符。以规范模式（ICANON）进行输入时识别此字符。当键入此字符时，等待被读的所有字节都立即传送给读进程。如果没有字节等待读，则返回0。在行首输入一个EOF符是向程序指示文件结束的正常方式。在以规范模式处理后，此字符即被删除，不送向读进程。

EOL　附加的行定界符，与NL作用相同。以规范模式（ICANON）进行输入时识别此字符，并将此字符返回给读进程。但通常不使用此字符。

EOL2　另一个行定界符，与NL作用相同。对此字符的处理方式与EOL字符相同。

ERASE　擦除字符（退格）。以规范模式（ICANON）输入时识别此字符。它擦除行中的前一个字符，但不会超越行首字符擦除上一行中的字符。在以规范模式处理后此字符即被删除，不送向读进程。

640

ERASE2　另一个擦除字符（退格）。对此字符的处理与ERASE完全相同。

INTR　中断字符。若设置了ISIG标志，则在输入中识别此字符。它产生SIGINT信号，该信号被送至前台进程组中的所有进程（参见图9-7）。在处理后，此字符即被删除，不送向读进程。

KILL　kill（杀死）字符。（名字“杀死”在这里又一次被误用，回忆kill函数，它将一信号发送给进程。此字符应被称为行擦除符，它与信号毫无关系。）以规范模式（ICANON）输入时识别此字符。它擦除一整行。在处理后，此字符即被删除，不送向读进程。

LNEXT　“字面上的下一个”字符（literal-next character）。以扩充方式（IEXTEN）输入时识别此字符，它使下一个字符的任何特殊含义都被忽略。这对本节提及的所有特殊字符都起作用。使用这一字符可向程序键入任何字符。在处理后，LNEXT字符即被删除，但输入的下一个字符则被传送给读进程。

NL　新行字符，它也被称为行定界符。不能更改此字符。以规范模式（ICANON）输入时识别此字符。此字符返回给读进程。

QUIT　退出字符。若设置了ISIG标志，则在输入中识别此字符。它产生SIGQUIT信号，该信号又被送至前台进程组中的所有进程（参见图9-7）。在处理后，此字符即被删除，不送向读进程。

回忆表10-1，INTR和QUIT的区别是：QUIT字符不仅按默认终止进程，而且也产生core文件。

REPRINT　再打印字符。以扩充规范模式（设置了IEXTEN和ICANON标志）进行输入时识别此字符。它使所有未读的输入被输出（再回显）。在处理后，此字符即被删除，不送向读进程。

START　启动字符。若设置了IXON标志则在输入中识别此字符；若设置IXOFF标志，则作为输出自动产生此字符。在IXON已设置时接收到的START字符使停止的输出（由以前输入的STOP字符造成）重新启动。在此情形下，此字符处理后即被删除，不送向读进程。

在IXOFF标志设置时，若输入不会使输入缓冲区溢出，则终端驱动程序自动地产生一START字符以恢复以前被停止的输入。

641

STATUS　BSD的状态-请求字符。以扩充规范模式（设置IEXTEN和ICANON标志）进行输入时识别此字符。它产生SIGINFO信号，该信号又被送至前台进程组中的所有进程（见图9-7）。另外，如果没有设置NOKERNINFO标志，则有关前台进程组的状态信息也显示在终端上。在处理后，此字符即被删除，不送向读进程。

STOP　停止字符。若设置了IXON标志，则在输入中识别此字符；若IXOFF标志已设置则作为输出自动产生此字符。在IXON已设置时接收到STOP字符则停止输出。在此情形下，处理后删除此字符，不送向读进程。当输入一个START字符后，停止的输出重新启动。

在IXOFF设置时，终端驱动程序自动地产生一个STOP字符以防止输入缓冲区溢出。

SUSP　挂起作业控制字符。若支持作业控制并且ISIG标志已设置，则在输入中识别此字符。它产生SIGTSTP信号，该信号又被送至前台进程组的所有进程（见图9-7）。在处理后，此字符即被删除，不送向读进程。

WERASE　字擦除字符。以扩充规范模式（设置IEXTEN和ICANON标志)进行输入时识别此字符。它擦除前一个字。首先，它向后跳过任一空白字符（空格或制表符），然后再向后越过前一记号，使光标处在前一个记号的第一个字符位置上。通常，前一个记号在碰到一个空白字符时即终止。但是，可用设置ALTWERASE标志来改变这一点。此标志使前一个记号在碰到第一个非字母、数字符时即终止。在处理后，此字符即被删除，不送向读进程。

需要为终端设备定义的另一个“字符”是BREAK。BREAK实际上并不是一个字符，而是在异步串行数据传送时发生的一个条件。依赖于串行接口，可以有多种方式通知设备驱动程序发生了BREAK条件。

> 大多数早期的串行终端有一个标记为BREAK的键，用它可以产生BREAK条件，这就使得很多人认为BREAK就是一个字符。某些较新的终端键盘没有BREAK键。在PC上，BREAK键有其他的用途。例如键入Ctrl+BREAK，可中断Windows命令解释器。

对于异步串行数据传送，BREAK是一个0值的位序列，其持续时间长于要求发送一个字节的时间。整个0值位序列被视为是一个BREAK。18.8节将说明如何用tcsendbreak函数发送一个BREAK。

642

18.4 获得和设置终端属性

使用函数tcgetattr和tcsetattr可以获得或设置termios结构。这样也就可以检测和修改各种终端选择标志和特殊字符，以使终端按我们所希望的方式进行操作。

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);

int tcsetattr(int filedes, int opt, const struct termios *termptr);
```

两个函数的返回值：若成功则返回0，若出错则返回-1

这两个函数都有一个指向termios结构的指针作为其参数，它们返回当前终端的属性，或者设置该终端的属性。因为这两个函数只对终端设备进行操作，所以若*filedes*并不引用一个终端设备则出错返回-1，errno设置为ENOTTY。

tcsetattr的参数*opt*使我们可以指定在什么时候新的终端属性才起作用。*opt*可以指定为下列常量中的一个：

- TCSANOW 更改立即发生。
- TCSADRAIN 发送了所有输出后更改才发生。若更改输出参数则应使用此选项。
- TCSAFLUSH 发送了所有输出后更改才发生。更进一步，在更改发生时未读的所有输入数据都被删除（刷清）。

tcsetattr函数的返回值易产生混淆。如果它执行了任意一种所要求的动作，即使未能执行所有要求的动作，它也返回0（表示成功）。如果该函数返回0，则我们有责任检查该函数是否执行了所有要求的动作。这就意味着，在调用tcsetattr设置所希望的属性后，需调用tcgetattr，然后将实际终端属性与所希望的属性相比较，以检测两者是否有区别。

18.5 终端选项标志

本节对表18-1至表18-4中列出的各个终端选项标志按字母顺序作进一步说明，指出该选项出现在四个终端标志字段中的哪一个（从选项名字中看不出它所处的字段），并说明该选项是否是Single UNIX Specification定义的，列出了支持该选项的平台。

列出的所有选项标志（除屏蔽标志外）都用一位或几位（设置或清除）表示，而屏蔽标志则定义多位，它们组合在一起，于是可以定义多个值。屏蔽标志有一个定义名，每个值也有一个名字。例如，为了设置字符长度，首先用字符长度屏蔽标志CSIZE将表示字符长度的位清0，然后设置下列值之一：CS5、CS6、CS7或CS8。

643

由Linux和Solaris支持的6个延迟值也有屏蔽标志：BSDLY、CRDLY、FFDLY、NLDLY、

TABDLY和VTDLY。欲了解每个延迟值的长度请参阅Solaris的termio(7I)手册页。如果指定了一个延迟，则OFILL和OFDEL标志决定是驱动器进行实际延迟还是只是传输填充字符。

实例

程序清单18-2例示了怎样使用屏蔽标志取或设置一个值。

程序清单18-2 tcgetattr和tcsetattr实例

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios  term;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 bits/byte\n");
        break;
    case CS6:
        printf("6 bits/byte\n");
        break;
    case CS7:
        printf("7 bits/byte\n");
        break;
    case CS8:
        printf("8 bits/byte\n");
        break;
    default:
        printf("unknown bits/byte\n");
    }

    term.c_cflag &= ~CSIZE;     /* zero out the bits */
    term.c_cflag |= CS8;        /* set 8 bits/byte */
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

644

下面说明各选项标志：

ALTWERASE (c_lflag，FreeBSD，Mac OS X) 此标志设置时，若输入了WERASE字符，则使用一个替换的字擦除算法。它不是向后移动到前一个空白字符为止，而是向后移动到第一个非字母、非数字字符为止。

BRKINT (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris) 若此标志设置，而IGNBRK未设置，则在接到BREAK时，刷清输入、输出队列，并产生一个SIGINT信号。如果此终端设备是一个控制终端，则将此信号送给前台进程组各进程。

如果IGNBRK和BRKINT都没有设置，但是设置了PARMRK，则BREAK被读作为三个字节序列\377，\0和\0，如果PARMRK也没有设置，则BREAK被读作为单个字符\0。

BSDLY (c_oflag, XSI, Linux, Solaris) 退格延迟屏蔽，此屏蔽的值是BS0或BS1。

CBAUDEXT (c_cflag, Solaris) 扩充的波特率。用于允许大于B38400的波特率。(18.7节将讨论波特率。)

CCAR_OFLOW (c_cflag, FreeBSD, Mac OS X) 打开输出的硬件流控制，该输出使用RS-232调制解调器载波信号（DCD，被称为数据-载波-检测）。这与早期的MDMBUF标志相同。

CCTS_OFLOW (c_cflag, FreeBSD, Mac OS X, Solaris) 使用Clear-To-Send (CTS) RS-232信号进行输出硬件的流控制。

CDSR_OFLOW (c_cflag, FreeBSD, Mac OS X) 按Data-Set-Ready (DSR) RS-232 信号进行输出流控制。

CDTR_IFLOW (c_cflag, FreeBSD, Mac OS X) 按Data-Terminal-Ready (DTR) RS-232信号进行输入流控制。

CIBAUDEXT (c_cflag, Solaris) 扩充的输入波特率，用于允许大于B38400的输入波特率。(18.7节将讨论波特率。)

CIGNORE (c_cflag, FreeBSD, Mac OS X) 忽略控制标志。

CLOCAL (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，则忽略调制解调器状态线。这通常意味着该设备是直接连接的。若此标志未设置，则打开一个终端设备常常会阻塞直到调制解调器回应呼叫并建立连接。

CMSPAR (c_oflag, Linux)选择标记或空奇偶校验。如果PARODD设置，则奇偶校验位总是1（标记奇偶校验）。否则奇偶校验位总是0（空奇偶校验）。

CRDLY (c_oflag, XSI, Linux, Solaris) 回车延迟屏蔽。此屏蔽的值是CR0、CR1、CR2和CR3。

645

CREAD (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，则接收装置被启用，可以接收字符。

CRTSCTS (c_cflag, FreeBSD, Linux, Mac OS X, Solaris) 其行为依赖于平台。对于Solaris，如果设置则允许输出硬件流控制。在另外三个平台上，允许输入、输出硬件流控制（等效于CCTS_OFLOW|CRTS_IFLOW)。

CRTS_IFLOW (c_cflag, FreeBSD, Mac OS X, Solaris) 输入的Request-To-Send (RTS) 流控制。

CRTSXOFF (c_cflag, Solaris) 如果设置，允许输入硬件流控制，Request-To-Send RS-232 信号状态控制了流控制。

CSIZE (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 此字段是一个屏蔽标志，它指明发送和接收的每个字节的位数。此长度不包括可能有的奇偶校验位。由此屏蔽标志定义的字段值是CS5、CS6、CS7和CS8，分别表示每个字节包含5、6、7和8位。

CSTOPB (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，则使用两位作为停止位，否则只使用一位作为停止位。

ECHO (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，则将输入字符回显（回送）到终端设备。在规范模式和非规范模式下都可

	以回显字符。
ECHOCTL	(c_lflag，FreeBSD，Linux，Mac OS X，Solaris)，如若设置并且ECHO也设置，则除ASCII TAB、ASCII NL、START和STOP字符外，其他ASCII控制符（ASCII字符集中的0～037）都被回显为^X，其中，X是相应控制字符代码值加8进制100所构成的字符。这就意味着ASCII Ctrl+A字符（8进制1）被回显为^A。ASCII DELETE字符（8进制177）则回显为^?。如若此标志未设置，则ASCII控制字符按其原样回显。如同ECHO标志，在规范模式和非规范模式下此标志对控制字符回显都起作用。 应当了解的是：某些系统回显EOF字符产生的作用有所不同，其原因是EOF的典型值是Ctrl+D，而这是ASCII EOT字符，它可能使某些终端挂断。请查看有关手册。
ECHOE	(c_lflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置并且ICANON也设置，则ERASE字符从显示中擦除当前行中的最后一个字符。这通常是在终端驱动程序中写三个字符序列“退格-空格-退格”而实现的。如若支持WERASE字符，则ECHOE用一个或若干个上述三字符序列擦除前一个字。 如若支持ECHOPRT标志，则在ECHOPRT标志没有设置的情况下，再采取这里所说明的对于ECHOE的动作。
ECHOK	(c_lflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置并且ICANON也设置，则KILL字符从显示中擦除当前行，或者输出NL字符(用以强调已擦除整个行)。 如若支持ECHOKE标志，则这里的说明假定ECHOKE标志没有设置。
ECHOKE	(c_lflag，FreeBSD，Linux，Mac OS X，Solaris）如若设置并且ICANON也设置，则回显KILL字符的方式是擦去行中的每一个字符。擦除每个字符的方法则由ECHOE和ECHOPRT标志选择。
ECHONL	(c_lflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置并且ICANON也设置，即使没有设置ECHO也回显NL字符。
ECHOPRT	(c_lflag，FreeBSD，Linux，Mac OS X，Solaris）如若设置并且ICANON和ECHO也都设置，则ERASE字符（以及WERASE字符，若受到支持）使所有正被擦除的字符按它们被擦除的方式打印。在硬拷贝终端上这常常是有用的，这样可以确切地看到哪些字符正被删除。
EXTPROC	(c_lflag，FreeBSD，Mac OS X）如若设置，规范字符处理在操作系统之外执行。如果串行通信外设卡执行某些行规程处理从而减轻主机处理器负载，或者使用伪终端（见第19章），那么就可作这种设置。
FFDLY	(c_oflag，XSI，Linux，Solaris）换页延迟屏蔽。此屏蔽标志值是FF0或FF1。
FLUSHO	(c_lflag，FreeBSD，Linux，Mac OS X，Solaris）如若设置，则刷清输出。当键入DISCARD字符时设置此标志；当键入另一个DISCARD字符时，此标志被清除。设置或清除此终端标志也可设置或清除此条件。
HUPCL	(c_cflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，

646

则当最后一个进程关闭此设备时，调制解调器控制线降至低电平（也就是调制解调器的连接断开）。

ICANON (c_lflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，则按规范模式工作（见18.10节）。这使下列字符起作用：EOF、EOL、EOL2、ERASE、KILL、REPRINT、STATUS和WERASE。输入字符被装配成行。如果不以规范模式工作，则读请求直接从输入队列取字符。在至少接到MIN个字节或已超过TIME值之前，read将不返回。详细情况见18.11节。

647

ICRNL (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置并且IGNCR未设置，则将接收到的CR字符转换成一个NL字符。

IEXTEN (c_lflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，则识别并处理扩充的、实现定义的特殊字符。

IGNBRK (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）在设置时，忽略输入中的BREAK条件。关于BREAK条件是产生SIGINT信号还是被读作为数据，请见BRKINT。

IGNCR (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，忽略接收到的CR字符。若此标志未设置而设置了ICRNL标志，则将接收到的CR字符转换成一个NL字符。

IGNPAR (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）在设置时，忽略带有结构错误（非BREAK）或奇偶错的输入字节。

IMAXBEL (c_iflag，FreeBSD，Linux，Mac OS X，Solaris）当输入队列满时响铃。

INLCR (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，则将接收到的NL字符转换成CR字符。

INPCK (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）当设置时，使输入奇偶校验起作用。如若未设置INPCK，则使输入奇偶校验不起作用。奇偶“产生和检测”和“输入奇偶性检验”是不同的两件事。奇偶位的产生和检测是由PARENB标志控制的。设置该标志后使串行接口的设备驱动程序对输出字符产生奇偶位，对输入字符则验证其奇偶性。标志PARODD决定该奇偶性应当是奇还是偶。如果一个到来的字符其奇偶性为错的，则检查INPCK标志的状态。若此标志已设置，则检查IGNPAR标志（以决定是否应忽略带奇偶错的输入字节），若不应忽略此输入字节，则检查PARMRK标志以决定向读进程应传送那些字符。

ISIG (c_lflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，则判别输入字符是否是要产生终端信号的特殊字符（INTR，QUIT，SUSP和DSUSP），若是，则产生相应信号。

ISTRIP (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）当设置时，有效输入字节被剥离为7位。当此标志未设置时，则保留全部8位。

648

IUCLC (c_iflag，Linux，Solaris）将输入的大写字符映射为小写字符。

IXANY (c_iflag，XSI，FreeBSD，Linux，Mac OS X，Solaris）使任一字符都能重新启动输出。

IXOFF (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，

则使启动-停止输入控制起作用。当终端驱动程序发现输入队列将要填满时，输出一个STOP字符。此字符应当由发送数据的设备识别，并使该设备暂停。此后，当已对输入队列中的字符进行了处理后，该终端驱动程序将输出一个START字符，使该设备恢复发送数据。

IXON (c_iflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，则使启动-停止输出控制起作用。当终端驱动程序接收到一个STOP字符时，输出暂停。在输出暂停时，下一个START字符恢复输出。如若未设置此标志，则START和STOP字符由进程读作为一般字符。

MDMBUF (c_cflag，FreeBSD，Mac OS X）按照调制解调器的载波标志进行输出流控制。这是CCAR_OFLOW标志的曾用名。

NLDLY (c_oflag，XSI，Linux，Solaris）换行延迟屏蔽。此屏蔽的值是NL0和NL1。

NOFLSH (c_lflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）按系统默认，当终端驱动程序产生SIGINT和SIGQUIT信号时，输入、输出队列都被刷新。另外，当它产生SIGSUSP信号时，输入队列被刷新。如若设置了NOFLSH标志，则在这些信号产生时，不对输入、输出队列进行刷新。

NOKERNINFO (c_lflag，FreeBSD，Mac OS X）当设置时，此标志防止STATUS字符把前台进程组的状态信息显示在终端上。但是不论本标志是否设置，STATUS字符使SIGINFO信号送至前台进程组中的所有进程。

OCRNL (c_oflag，XSI，FreeBSD，Linux，Solaris）如若设置，将输出的CR字符映射为NL。

OFDEL (c_oflag，XSI，Linux，Solaris）如若设置，则输出填充字符是ASCII DEL，否则它是ASCII NUL。参见OFILL标志。

OFILL (c_oflag，XSI，Linux，Solaris）如若设置，则为了实现延迟，发送填充字符（ASCII DEL或ASCII NUL，参见OFDEL标志)，而不使用时间延迟。参见如下6个延迟屏蔽标志：BSDLY，CRDLY，FFDLY，NLDLY，TABDLY以及VTDLY。

OLCUC (c_oflag，Linux，Solaris）如若设置，将输出的小写字符映射为大写。 649

ONLCR (c_oflag，XSI，FreeBSD，Linux，Mac OS X，Solaris）如若设置，将输出的NL字符映射为CR-NL。

ONLRET (c_oflag，XSI，FreeBSD，Linux，Solaris）如若设置，则输出的NL字符应该执行回车功能。

ONOCR (c_oflag，XSI，FreeBSD，Linux，Solaris）如若设置，则在0列不输出CR。

ONOEOT (c_oflag，FreeBSD，Mac OS X）如若设置，则在输出中删除EOT字符（^D）。在将Ctrl+D解释为挂断的终端上这可能是需要的。

OPOST (c_oflag，POSIX.1，FreeBSD，Linux，Mac OS X，Solaris）如若设置，则进行实现定义的输出处理。关于c_oflag字的各种实现定义的标志，见表18-4。

OXTABS (c_oflag，FreeBSD，Mac OS X）如若设置，制表符在输出中被扩展为空格。这与将水平制表延迟（TABDLY）设置为XTABS或TAB3产生的效果相同。

PARENB (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，则对输出字符产生奇偶位，对输入字符则执行奇偶性检验。若PARODD已设置，则奇偶校验是奇校验，否则是偶校验。参见对INPCK、IGNPAR和PARMRK标志的讨论。

PAREXT (c_cflag, Solaris) 选择标记或空奇偶性。若PARODD设置，则奇偶位总是1（标记奇偶性）；否则奇偶位总是0（空奇偶性）。

PARMRK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 当设置时，如果IGNPAR未设置，则结构性错（非BREAK）或奇偶错的字节由进程读作为三个字符序列\377，\0和*X*，其中*X*是接收到的具有错误的字节。如若ISTRIP未设置，则一个有效的\377被传送给进程时为\377,\377。如若IGNPAR和PARMRK都未设置，则结构性错或奇偶错的字节都被读作为一个字符\0。

PARODD (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，则输出和输入字符的奇偶性都是奇，否则为偶。注意，PARENB标志控制奇偶性的产生和检测。

当CMSPAR或PAREXT标志设置时，PARODD标志也控制是否使用标记或空格的奇偶性。

PENDIN (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，则在下一个字符输入时，尚未读的任何输入都由系统重新打印。这一动作与键入REPRINT字符时的作用相类似。

650

TABDLY (c_oflag, XSI, Linux, Solaris) 水平制表延迟屏蔽标志。此屏蔽标志的值是TAB0、TAB1、TAB2或TAB3。

XTABS的值等于TAB3。此值使系统将制表符扩展成空格。系统假定制表符在屏幕上每8个空格处设置一个。我们不能更改此假定。

TOSTOP (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) 如若设置，并且该实现支持作业控制，则将信号SIGTTOU送到试图写控制终端的一个后台进程的进程组。按默认，此信号暂停该进程组中所有进程。如果写控制终端的进程忽略或阻塞此信号，则终端驱动程序不产生此信号。

VTDLY (c_oflag, XSI, Linux, Solaris) 垂直制表延迟屏蔽标志。此屏蔽标志的值是VT0或VT1。

XCASE (c_lflag, Linux, Solaris) 如若设置，并且ICANON也设置，则认为终端只是大写终端，所有输入都变换为小写。为了输入一个大写字符，须在其前加一个\。与之类似，输出一个大写字符也在其前加一个\。（这一标志如今已经过时，现在几乎所有终端都支持大、小写字符。）

18.6 stty命令

上节说明的所有选项，在程序中都可用tcgetattr和tcsetattr函数（见18.4节）进行检查和更改。在命令行（或shell脚本）中则可用stty(1)命令进行检查和更改。stty(1)命令是表18-5中所列的前6个函数的接口。如果以-a选项执行此命令，则显示终端的所有选项：

```
$ stty -a
speed 9600 baud; 25 rows; 80 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -ocrnl -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts
        -dsrflow -dtrflow -mdmbuf
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; erase2 = ^?; intr = ^C; kill = ^U;
        lnext = ^V; min = 1; quit = ^; reprint = ^R; start = ^Q;
        status = ^T; stop = ^S; susp = ^Z; time = 0; werase = ^W;
```

651

选项名前若有连字符，则表示该选项禁用。最后四行显示各终端特殊字符的当前设置（见18.3节）。第1行显示当前终端窗口的行数和列数，18.12节将对此进行讨论。

stty使用标准输入获得和设置终端的选项标志。虽然，某些较早的实现使用标准输出，但POSIX.1要求使用标准输入。本书讨论的四种实现都提供了在标准输入上操作的stty版本。这意味着如果我希望了解名为tty1a终端的设置，那么我们可以键入

```
stty -a </dev/tty1a
```

18.7 波特率函数

波特率（baud rate）是一个以往采用的术语，现在它指的是“位/秒”（bits per second）。虽然大多数终端设备对输入和输出使用同一波特率，但是只要硬件许可，可以将它们设置为两个不同值。

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);

speed_t cfgetospeed(const struct termios *termptr);

                                        两个函数的返回值：波特率值

int cfsetispeed(struct termios *termptr, speed_t speed);

int cfsetospeed(struct termios *termptr, speed_t speed);

                        两个函数的返回值：若成功则返回0，若出错则返回-1
```

两个cfget函数的返回值，以及两个cfset函数的*speed*参数都是下列常量之一：B50、B75、B110、B134、B150、B200、B300、B600、B1200、B1800、B2400、B4800、B9600、B19200或B38400。常量B0表示“挂断”。在调用tcsetattr时，如若将输出波特率指定为B0，则调制解调器的控制线就不再起作用。

大多数系统定义了另外的波特率值，例如B57600以及B115200。

使用这些函数时，应当理解输入、输出波特率是存放在图18-3所示的设备termios结构中的。在调用任一cfget函数之前，先要用tcgetattr获得设备的termios结构。与此类似，在调用

任一cfset函数后，应将波特率设置到termios结构中。为使这种更改影响到设备，应当调用tcsetattr函数。如果所设置的波特率有错，则在调用tcsetattr之前，不会发现这种错误。

652

这4个波特率函数使应用程序不必考虑具体实现在termios结构中表示波特率的不同方法。BSD派生的平台趋向于存放波特率的数值（例如9600波特就存放为9600），同时Linux和系统V派生的平台趋向于以位屏蔽方式表示波特率。从cfget函数得到的以及向cfset传送的速度值与它们存放在termios结构中的一样。

18.8 行控制函数

下列4个函数提供了终端设备的行控制能力。其中，参数*filedes*引用一个终端设备，否则出错返回，errno设置为ENOTTY。

```
#include <termios.h>

int tcdrain(int filedes);

int tcflow(int filedes, int action);

int tcflush(int filedes, int queue);

int tcsendbreak(int filedes, int duration);
```

四个函数返回值：若成功则返回0，若出错则返回-1

tcdrain函数等待所有输出都被发送。tcflow用于对输入和输出流控制进行控制。*action*参数应当是下列四个值之一。

TCOOFF 输出被挂起。

TCOON 重新启动以前被挂起的输出。

TCIOFF 系统发送一个STOP字符。这将使终端设备暂停发送数据。

TCION 系统发送一个START字符。这将使终端恢复发送数据。

tcflush函数刷清（抛弃）输入缓冲区或输出缓冲区。输入缓冲区中的数据是终端驱动程序已接收到，但用户程序尚未读的；输出缓冲区中的数据是用户程序已经写，但尚未发送的。*queue*参数应当是下列三个常量之一：

TCIFLUSH 刷清输入队列。

TCOFLUSH 刷清输出队列。

653

TCIOFLUSH 刷清输入、输出队列。

tcsendbreak函数在一个指定的时间区间内发送连续的0位流。若*duration*参数为0，则此种发送延续0.25至0.5秒之间。POSIX.1说明若*duration*非0，则发送时间依赖于实现。

18.9 终端标识

历史沿袭至今，在大多数UNIX系统中，控制终端的名字是/dev/tty。POSIX.1提供了一个运行时函数，可被用来确定控制终端的名字。

```
#include <stdio.h>

char *ctermid(char *ptr);
```

返回值：若成功则返回指向控制终端名的指针，若出错则返回指向空字符串的指针

如果*ptr*非null，则它被认为是一个指针，指向长度至少为L_ctermid字节的数组，进程的控制终端名存放在该数组中。常量L_ctermid定义在<stdio.h>中。若*ptr*是一个空指针，则该函数为数组（通常作为静态变量）分配空间。同样，进程的控制终端名存放在该数组中。

在这两种情况中，该数组的起始地址被作为函数值返回。因为大多数UNIX系统都使用/dev/tty作为控制终端名，所以此函数的主要作用是帮助提高向其他操作系统的可移植性。

当调用ctermid函数时，本书说明的所有四种平台都返回字符串/dev/tty。

实例：ctermid函数

程序清单18-3是POSIX.1 ctermid函数的一个实现。

程序清单18-3　POSIX.1 ctermid函数的实现

```
#include    <stdio.h>
#include    <string.h>

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty"));    /* strcpy() returns str */
}
```

654

注意，因为我们无法确定调用者的缓冲区大小，所以也就不能防止过度使用该缓冲区。□

另外两个与终端标识有关的函数是isatty和ttyname。前者在文件描述符引用一个终端设备时返回真，而后者则返回在该文件描述符上打开的终端设备的路径名。

```
#include <unistd.h>

int isatty(int filedes);

                    返回值：若为终端设备则返回1（真），否则返回0（假）

char *ttyname(int filedes);

                    返回值：指向终端路径名的指针，若出错则返回NULL
```

实例：isatty函数

如程序清单18-4所示，isatty函数很容易实现。其中只使用了一个终端专用的函数tcgetattr（如果成功执行，它不改变任何东西），并取其返回值。

程序清单18-4　POSIX.1 isatty函数的实现

```
#include    <termios.h>

int
isatty(int fd)
{
    struct termios  ts;
```

```
    return(tcgetattr(fd, &ts) != -1); /* true if no error (is a tty) */
}
```

用程序清单18-5中的程序测试isatty函数。

程序清单18-5　测试isatty函数

```
#include "apue.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "not a tty");
    exit(0);
}
```

655

当运行程序清单18-5中的程序时，我们可以得到下面的结果：

```
$ ./a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ ./a.out </etc/passwd 2>/dev/null
fd 0: not a tty
fd 1: tty
fd 2: not a tty
```

□

实例：ttyname函数

ttyname函数（见程序清单18-6）比较长，因为它要搜索所有设备表项，寻找匹配项。

程序清单18-6　POSIX.1 ttyname函数的实现

```
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    <string.h>
#include    <termios.h>
#include    <unistd.h>
#include    <stdlib.h>

struct devdir {
    struct devdir   *d_next;
    char            *d_name;
};

static struct devdir    *head;
static struct devdir    *tail;
static char             pathname[_POSIX_PATH_MAX + 1];

static void
add(char *dirname)
{
    struct devdir   *ddp;
    int             len;

    len = strlen(dirname);

    /*
```

```
     * Skip ., .., and /dev/fd.
     */
    if ((dirname[len-1] == '.') && (dirname[len-2] == '/' ||
      (dirname[len-2] == '.' && dirname[len-3] == '/')))
        return;
    if (strcmp(dirname, "/dev/fd") == 0)
        return;
    ddp = malloc(sizeof(struct devdir));
    if (ddp == NULL)
        return;

    ddp->d_name = strdup(dirname);
    if (ddp->d_name == NULL) {
        free(ddp);
        return;
    }
    ddp->d_next = NULL;
    if (tail == NULL) {
        head = ddp;
        tail = ddp;
    } else {
        tail->d_next = ddp;
        tail = ddp;
    }
}

static void
cleanup(void)
{
    struct devdir   *ddp, *nddp;

    ddp = head;
    while (ddp != NULL) {
        nddp = ddp->d_next;
        free(ddp->d_name);
        free(ddp);
        ddp = nddp;
    }
    head = NULL;
    tail = NULL;
}

static char *
searchdir(char *dirname, struct stat *fdstatp)
{
    struct stat     devstat;
    DIR             *dp;
    int             devlen;
    struct dirent   *dirp;

    strcpy(pathname, dirname);
    if ((dp = opendir(dirname)) == NULL)
        return(NULL);
    strcat(pathname, "/");
    devlen = strlen(pathname);
    while ((dirp = readdir(dp)) != NULL) {
        strncpy(pathname + devlen, dirp->d_name,
          _POSIX_PATH_MAX - devlen);

        /*
         * Skip aliases.
         */
```

656

657

```
        if (strcmp(pathname, "/dev/stdin") == 0 ||
          strcmp(pathname, "/dev/stdout") == 0 ||
          strcmp(pathname, "/dev/stderr") == 0)
            continue;
        if (stat(pathname, &devstat) < 0)
            continue;
        if (S_ISDIR(devstat.st_mode)) {
            add(pathname);
            continue;
        }
        if (devstat.st_ino == fdstatp->st_ino &&
          devstat.st_dev == fdstatp->st_dev) {  /* found a match */
            closedir(dp);
            return(pathname);
        }
    }

    closedir(dp);
    return(NULL);
}

char *
ttyname(int fd)
{
    struct stat     fdstat;
    struct devdir   *ddp;
    char            *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    rval = searchdir("/dev", &fdstat);
    if (rval == NULL) {
        for (ddp = head; ddp != NULL; ddp = ddp->d_next)
            if ((rval = searchdir(ddp->d_name, &fdstat)) != NULL)
                break;
    }

    cleanup();
    return(rval);
}
```

此处用到的方法是读/dev目录，寻找具有相同设备号和i节点编号的表项。回忆4.23节，每个文件系统有一个唯一的设备号（stat结构中的st_dev字段，见4.2节），文件系统中的每个目录项有一个唯一的i节点号（stat结构中的st_ino字段）。在此函数中假定当找到一个匹配的设备号和匹配的i节点号时，就找到了所希望的目录项。也可验证这两个表项与st_rdev字段（终端设备的主、次设备号）相匹配，以及该目录项是一个字符特殊文件。但是，因为已经验证了文件描述符参数是一个终端设备以及一个字符特殊设备，而且在UNIX系统中，匹配的设备号和i节点号是唯一的，所以不再需要作另外的比较。

658

我们的终端名可能在/dev的子目录中。于是，需要搜索在/dev之下的整个文件系统子树。我们跳过了很多会产生不正确或奇怪结果的目录，它们是/dev/.、/dev/..和/dev/fd。我们也跳过了一些别名，即/dev/stdin、/dev/stdout以及/dev/stderr，它们是对在/dev/fd目录中文件的符号链接。

用程序清单18-7测试这一实现。

程序清单18-7 测试ttyname函数

```
#include "apue.h"

int
main(void)
{
    char *name;

    if (isatty(0)) {
        name = ttyname(0);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 0: %s\n", name);
    if (isatty(1)) {
        name = ttyname(1);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 1: %s\n", name);
    if (isatty(2)) {
        name = ttyname(2);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 2: %s\n", name);
    exit(0);
}
```

659

运行该程序得到

```
$ ./a.out < /dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttyp3
fd 2: not a tty
```

□

18.10 规范模式

规范模式很简单：发一个读请求，输入完一行后，终端驱动程序即返回。下列几个条件都会造成读返回。

- 所要求的字节数已读到时，读返回。无需读一个完整的行。如果读了部分行，那么也不会丢失任何信息，下一次读从前一次读的停止处开始。
- 当读到一个行定界符时，读返回。回忆18.3节，在规范模式中下列字符被解释为“行结束”：NL、EOL、EOL2和EOF。另外，在18.5节中也曾说明，如若已设置ICRNL，但未设置IGNCR，则CR字符的作用与NL字符一样，所以它也终止一行。

  在这5个行定界符中，其中只有一个EOF字符在终端驱动程序对其进行处理后即被删除。其他4个字符则作为该行的最后一个字符返回给调用者。

• 如果捕捉到信号而且该函数并不自动重启动（见10.5节），则读也返回。

实例：getpass函数

下面说明getpass函数，它读入用户在终端上键入的口令。此函数由UNIX login(1)和crypt(1)程序调用。为了读口令，该函数必须禁止回显，但仍可使终端以规范模式进行工作，因为用户在键入口令后，一定要键入回车，这样也就构成了一个完整行。程序清单18-8是一个典型的UNIX实现。

程序清单18-8　getpass函数的实现

```
#include    <signal.h>
#include    <stdio.h>
#include    <termios.h>

#define MAX_PASS_LEN    8       /* max #chars for user to enter */

char *
getpass(const char *prompt)
{
    static char     buf[MAX_PASS_LEN + 1];  /* null byte at end */
    char            *ptr;
    sigset_t        sig, osig;
    struct termios  ts, ots;
    FILE            *fp;
    int             c;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig);
    sigaddset(&sig, SIGINT);        /* block SIGINT */
    sigaddset(&sig, SIGTSTP);       /* block SIGTSTP */
    sigprocmask(SIG_BLOCK, &sig, &osig);    /* and save mask */

    tcgetattr(fileno(fp), &ts);     /* save tty state */
    ots = ts;                       /* structure copy */
    ts.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &ts);
    fputs(prompt, fp);

    ptr = buf;
    while ((c = getc(fp)) != EOF && c != '\n')
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    *ptr = 0;               /* null terminate */
    putc('\n', fp);         /* we echo a newline */

    tcsetattr(fileno(fp), TCSAFLUSH, &ots); /* restore TTY state */
    sigprocmask(SIG_SETMASK, &osig, NULL);  /* restore mask */
    fclose(fp);             /* done with /dev/tty */
    return(buf);
}
```

在此例中，有很多方面应当考虑：

• 调用ctermid函数打开控制终端，而不是直接将/dev/tty写在程序中。

• 只是读、写控制终端，如果不能以读、写模式打开此设备则出错返回。在有些系统中也使用一些其他约定。在BSD中，如果不能以读、写模式打开控制终端，则getpass从标

准输入读，写到标准出错文件中。系统V版本则总是写到标准出错文件中，但只从控制终端读。

- 阻塞两个信号SIGINT和SIGTSTP。如果不这样做，则在输入INTR字符时就会使程序终止，并使终端仍处于禁止回显状态。与此相关似，输入SUSP字符时将使程序暂停，并且在禁止回显状态下返回到shell。在禁止回显时，选择了阻塞这两个信号。在读口令期间如果发生了这两个信号，则它们被保持，直到getpass返回前才解除对它们的阻塞。也有其他方法来处理这些信号。某些getpass版本忽略SIGINT（保存它以前的动作），在返回前则将其动作恢复为以前的值。这就意味着在该信号被忽略期间所发生的这种信号都丢失。其他版本捕捉SIGINT（保存它以前的动作），如果捕捉到此信号，则在复置终端状态和信号动作后，用kill函数发送此信号。没有一个getpass版本捕捉、忽略或阻塞SIGQUIT，所以键入QUIT字符就会使程序夭折，并且终端极可能仍处于禁止回显状态。
- 请注意，某些shell（例如Korn shell）在以交互方式读输入时都使终端处于回显状态。这些shell是提供命令行编辑的shell，因此在每次输入一条交互命令时都处理终端状态。所以如果在这种shell下调用此程序，并且用QUIT字符使其夭折，则这种shell可以恢复回显状态。不提供命令行编辑的shell（例如Bourne shell）将使程序夭折，并使终端仍处于不回显状态。如果对终端做了这种操作，则stty命令能使终端回复到回显状态。
- 我们使用标准I/O读、写控制终端。我们特地将流设置为不带缓冲的，否则在流的读、写之间可能会有某些相互作用（这样就需调用fflush）。也可使用不带缓冲的I/O（见第3章），但是在这种情况下就要用read来实现getc。
- 最多只存储8个字符作为口令。输入的多余字符则被忽略。

660
~
661

程序清单18-9调用getpass并且打印出我们的输入。这是为了验证ERASE和KILL字符是否正常工作（如同它们在规范模式下所应该的那样）。

程序清单18-9 调用getpass函数

```
#include "apue.h"

char    *getpass(const char *);

int
main(void)
{
    char    *ptr;

    if ((ptr = getpass("Enter password:")) == NULL)
        err_sys("getpass error");
    printf("password: %s\n", ptr);

    /* now use password (probably encrypt it) ... */

    while (*ptr != 0)
        *ptr++ = 0;     /* zero it out when we're done with it */
    exit(0);
}
```

662

调用getpass函数的程序完成后，为了安全起见，应清除存放过用户键入的未经加密的明文口令的存储区。如果该程序会产生其他用户能读的core文件，或者如果某个其他进程能够设法读该进程的存储空间，则它们就能读到这种明文口令。（我们用“明文”表示在getpass输出的提示符后输入的密码。多数UNIX系统会把此明文密码改成“加密”密码。例如，密码

文件中的pw_passwd字段包含的就是加密密码，而不是明文密码。） □

18.11 非规范模式

关闭termios结构中c_lflag字段的ICANON标志就使终端处于非规范模式。在非规范模式中，输入数据并不组成行，不处理下列特殊字符（参见18.3节）：ERASE、KILL、EOF、NL、EOL、EOL2、CR、REPRINT、STATUS和WERASE。

如前所述，规范模式很容易：系统每次返回一行。但在非规范模式下，系统怎样才能知道在什么时候将数据返回给我们呢？如果它一次返回一个字节，那么系统开销就很大。（回忆表3-2，从中可以看到每次读一个字节的开销会有多大。如果每次返回的数据加倍，系统调用的开销就可减半。）在启动读数据之前，往往不知道要读多少数据，所以系统不能总是一次返回多个字节。

解决方法是：当已读了指定量的数据后，或者已经过了给定的时间后，即通知系统返回。这种技术使用了termios结构中c_cc数组的两个变量：MIN和TIME。c_cc数组中的这两个元素的下标名为VMIN和VTIME。

MIN说明一个read返回前的最小字节数。TIME说明等待数据到达的分秒数（秒的1/10为分秒）。有下列四种情形：

情形A：MIN > 0，TIME > 0

TIME说明字节间的计时器，在接到第一个字节时才启动它。在该计时器超时之前，若已接到MIN个字节，则read返回MIN个字节。如果在接到MIN个字节之前，该计时器已超时，则read返回已接收到的字节（因为只有在接到第一个字节时才启动，所以在计时器超时时，至少返回了1个字节）。这种情形中，在接到第一个字节之前，调用者阻塞。如果在调用read时数据已经可用，则这如同在read后数据立即被接收到一样。

情形B：MIN > 0，TIME == 0

663 已经接到了MIN个字节时，read才返回。这可以造成read无限期地阻塞。

情形C：MIN == 0，TIME > 0

TIME指定了一个调用read时启动的读计时器。（与情形A相比较，两者是不同的。在情形A中，非0的TIME表示字节间的计时器，在接到第一个字节时才启动它。）在接到1个字节或者该计时器超时时，read即返回。如果是计时器超时，则read返回0。

情形D：MIN == 0，TIME == 0

如果有数据可用，则read最多返回所要求的字节数。如果无数据可用，则read立即返回0。

在所有这些情形中，MIN只是最小值。如果程序要求的数据多于MIN个字节，那么它可能接收到所要求的字节数。这也适用于MIN为0的情形C和D。

表18-7摘要列出了非规范模式输入的四种不同情形。在图中，*nbytes*是read的第三个参数（返回的最大字节数）。

POSIX.1允许下标VMIN和VTIME的值分别与VEOF和VEOL相同。确实，Solaris就是这样做的。这样就提供了与系统V早期版本的向上兼容性。但是，这也带来了可移植性问题。从非规范模式转换为规范模式时，必须恢复VEOF和VEOL，如果不这样做，那么VMIN等于VEOF，并且它已被设置为典型值1，于是文件结束字符就变成Ctrl+A。解决这一问题最简单的方法是：在转入非规范模式时将整个termios结构保存起来。在以后再转回规范模式时恢复它。

表18-7 非规范输入的四种情形

	MIN>0	MIN==0
TIME > 0	A：在计时器超时前，read返回[MIN, *nbytes*]；若计时器超时，read返回[1, MIN]（TIME=字节间计时器，调用者可能无限阻塞）	C：在计时器超时前，read返回[1, *nbytes*]；若计时器超时，read返回0（TIME=read计时器）
TIME == 0	B：可用时，read返回[MIN, *nbytes*]（调用者可能无限阻塞）	D：read立即返回[0, *nbytes*]

程序清单18-10定义了函数tty_cbreak和tty_raw，它们将终端分别设置为cbreak模式和raw（原始）模式（术语cbreak和raw来自于V7的终端驱动程序）。tty_reset函数的功能是将终端恢复为以前的工作模式（也就是调用tty_cbreak和tty_raw之前的工作模式）。

如果已调用tty_cbreak，那么在调用tty_raw之前需要调用tty_reset。如果已调用tty_raw，然后又要调用tty_cbreak，那么在此之前同样也要调用tty_reset。这减少了出错时终端处于不可用状态的机会。

664

该程序还提供了tty_atexit和tty_termios两个函数。tty_atexit可被登记为终止处理程序，以保证exit恢复终端工作模式；tty_termios则返回一个指向原先的规范模式termios结构的指针。

程序清单18-10 将终端模式设置为原始或cbreak模式

```
#include "apue.h"
#include <termios.h>
#include <errno.h>

static struct termios       save_termios;
static int                  ttysavefd = -1;
static enum { RESET, RAW, CBREAK }  ttystate = RESET;

int
tty_cbreak(int fd)  /* put terminal into a cbreak mode */
{
    int             err;
    struct termios  buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* structure copy */

    /*
     * Echo off, canonical mode off.
     */
    buf.c_lflag &= ~(ECHO | ICANON);

    /*
     * Case B: 1 byte at a time, no timer.
```

```
     */
    buf.c_cc[VMIN] = 1;
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);

    /*
     * Verify that the changes stuck.  tcsetattr can return 0 on
     * partial success.
     */
    if (tcgetattr(fd, &buf) < 0) {
        err = errno;
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = err;
        return(-1);
    }
    if ((buf.c_lflag & (ECHO | ICANON)) || buf.c_cc[VMIN] != 1 ||
      buf.c_cc[VTIME] != 0) {
        /*
         * Only some of the changes were made.  Restore the
         * original settings.
         */
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = EINVAL;
        return(-1);
    }

    ttystate = CBREAK;
    ttysavefd = fd;
    return(0);
}

int
tty_raw(int fd)     /* put terminal into a raw mode */
{
    int             err;
    struct termios  buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* structure copy */

    /*
     * Echo off, canonical mode off, extended input
     * processing off, signal chars off.
     */
    buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    /*
     * No SIGINT on BREAK, CR-to-NL off, input parity
     * check off, don't strip 8th bit on input, output
     * flow control off.
     */
    buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    /*
     * Clear size bits, parity checking off.
     */
```

665

```
    buf.c_cflag &= ~(CSIZE | PARENB);

    /*
     * Set 8 bits/char.
     */
    buf.c_cflag |= CS8;

    /*
     * Output processing off.
     */
    buf.c_oflag &= ~(OPOST);

    /*
     * Case B: 1 byte at a time, no timer.
     */
    buf.c_cc[VMIN] = 1;
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);

    /*
     * Verify that the changes stuck.  tcsetattr can return 0 on
     * partial success.
     */
    if (tcgetattr(fd, &buf) < 0) {
        err = errno;
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = err;
        return(-1);
    }
    if ((buf.c_lflag & (ECHO | ICANON | IEXTEN | ISIG)) ||
      (buf.c_iflag & (BRKINT | ICRNL | INPCK | ISTRIP | IXON)) ||
      (buf.c_cflag & (CSIZE | PARENB | CS8)) != CS8 ||
      (buf.c_oflag & OPOST) || buf.c_cc[VMIN] != 1 ||
      buf.c_cc[VTIME] != 0) {
        /*
         * Only some of the changes were made.  Restore the
         * original settings.
         */
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = EINVAL;
        return(-1);
    }

    ttystate = RAW;
    ttysavefd = fd;
    return(0);
}

int
tty_reset(int fd)       /* restore terminal's mode */
{
    if (ttystate == RESET)
        return(0);
    if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
        return(-1);
    ttystate = RESET;
    return(0);
}

void
tty_atexit(void)        /* can be set up by atexit(tty_atexit) */
```

666

667

```
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}

struct termios *
tty_termios(void)       /* let caller see original tty state */
{
    return(&save_termios);
}
```

我们对*cbreak*模式的定义如下：

- 非规范模式。如本节开始处所述，这种模式不对某些输入特殊字符进行处理。这种模式没有关闭对信号的处理，所以用户可以键入任一终端产生的信号。调用程序应当捕捉这些信号，否则这种信号就可能终止程序，并且终端仍将处于cbreak模式。

  作为一般规则，在编写更改终端模式的程序时，应当捕捉大多数信号，以便在程序终止前恢复终端模式。
- 关闭回显（ECHO）标志。
- 每次输入一个字节。为此将MIN设置为1，将TIME设置为0。这是表18-7中的情形B。至少有一个字节可用时，read再返回。

对原始模式的定义如下：

- 非规范模式。也关闭了对产生信号字符（ISIG）和扩充输入字符（IEXTEN）的处理。另外，禁用BRKINT，这样就使BREAK字符不再产生信号。
- 关闭回显（ECHO）标志。
- 禁用ICRNL、INPCK、ISTRIP和IXON标志。从而不再将输入的CR字符变换为NL（ICRNL），使输入奇偶校验不起作用（INPCK），不再剥离输入字节的第8位（ISTRIP），不进行输出流控制（IXON）。
- 8位字符（CS8），且禁用奇偶性检测（PARENB）。
- 禁用所有输出处理（OPOST）。
- 每次输入一个字节（MIN = 1，TIME = 0）。

668

程序清单18-11测试原始模式和cbreak模式。

程序清单18-11 测试原始终端模式和cbreak终端模式

```
#include "apue.h"

static void
sig_catch(int signo)
{
    printf("signal caught\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

int
main(void)
{
    int     i;
    char    c;

    if (signal(SIGINT, sig_catch) == SIG_ERR)   /* catch signals */
        err_sys("signal(SIGINT) error");
```

```
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("signal(SIGTERM) error");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("Enter raw mode characters, terminate with DELETE\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177)     /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");
    if (tty_cbreak(STDIN_FILENO) < 0)
        err_sys("tty_cbreak error");
    printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        c &= 255;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");

    exit(0);
}
```

669

运行该程序可以观察这两种终端工作模式的工作情况。

```
$ ./a.out
Enter raw mode characters, terminate with DELETE
                                                 4
                                                  33
                                                    133
                                                       61
                                                         70
                                                           176
                                键入DELETE
Enter cbreak mode characters, terminate with SIGINT
1                               键入Ctrl+A
10                              键入退格
signal caught                   键入中断符
```

在原始模式中，输入的字符是Ctrl+D(04)和特殊功能键F7。在所用的终端上，此功能键产生5个字符：*ESC* (033)，*[* (0133)，*1* (061)，*8* (070)和~ (0176)。注意，在原始模式下关闭了输出处理(~OPOST)，所以在每个字符后没有得到回车符。另外也要注意的是，在cbreak模式下，不对输入特殊字符进行处理（所以对Ctrl+D、文件结束符和退格等不进行特殊处理），但是对终端产生的信号则进行处理。 □

18.12 终端的窗口大小

大多数UNIX系统都提供了一种功能，可以对当前终端窗口的大小进行跟踪，在窗口大小

发生变化时，使内核通知前台进程组。内核为每个终端和伪终端保存一个winsize结构：

```
struct winsize {
  unsigned short  ws_row;     /* rows, in characters */
  unsigned short  ws_col;     /* columns, in characters */
  unsigned short  ws_xpixel;  /* horizontal size, pixels (unused) */
  unsigned short  ws_ypixel;  /* vertical size, pixels (unused) */
};
```

此结构的作用如下：

- 用ioctl（见3.15节）的TIOCGWINSZ命令可以取此结构的当前值。
- 用ioctl的TIOCSWINSZ命令可以将此结构的新值存放到内核中。如果此新值与存放在内核中的当前值不同，则向前台进程组发送SIGWINCH信号。（注意，从表10-1中可以看出，此信号的系统默认动作是忽略。）

670

- 除了存放此结构的当前值以及在此值改变时产生一个信号以外，内核对该结构不进行任何其他操作。对结构中的值进行解释完全是应用程序的工作。
- 提供这种功能的目的是，当窗口大小发生变化时通知应用程序（例如vi编辑程序）。应用程序接到此信号后，它可以取窗口大小的新值，然后重绘屏幕。

程序清单18-12打印当前窗口大小，然后休眠。每次窗口大小改变时，就捕捉到SIGWINCH信号，然后打印新的窗口大小。必须用一个信号终止此程序。

程序清单18-12　打印窗口大小

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

static void
pr_winsize(int fd)
{
    struct winsize  size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
}

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);
    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("signal error");
```

```
    pr_winsize(STDIN_FILENO);    /* print initial size */
    for ( ; ; )                  /* and sleep forever */
        pause();
}
```

671

在一个带窗口终端的系统上运行此程序得到：

```
$ ./a.out
35 rows, 80 columns             起始长度
SIGWINCH received               更改窗口大小：捕捉到信号
40 rows, 123 columns
SIGWINCH received               再来一次
42 rows, 33 columns
^? $                            击中断键以终止
```

□

18.13 termcap，terminfo和curses

termcap的意思是终端能力（terminal capability），它指的是文本文件/etc/termcap和一套读此文件的例程。termcap这种技术是在伯克利发展起来的，主要是为了支持vi编辑器。termcap文件包含了对各种终端的说明：终端支持哪些功能（行数、列数、是否支持退格等），如何使终端执行某些操作（清屏、将光标移动到指定位置等）。把这些信息从编译过的程序中取出来并把它们放在易于编辑的文本文件中，这样就使得vi能在很多不同的终端上运行。

后来，支持termcap文件的一套例程从vi编辑程序中抽取出来，放在一个单独的curses库中。为使这套库被要进行屏幕处理的任何程序使用，人们给它增加了很多功能。

termcap这种技术不是很完善。当越来越多类型的终端被加到该数据文件中时，为了找到一个特定的终端就须使用较长的时间扫描此文件。此数据文件也只用两个字符的名字来标识不同的终端属性。这些缺陷导致开发另一种新技术——terminfo及与其相关的curses库。在terminfo中的终端说明基本上是文本说明的编译版本，在运行时易于快速定位。terminfo从SVR2就开始使用，此后所有的系统V版本都使用了它。

> 以往，基于系统V的系统使用terminfo，而BSD派生的系统则使用termcap，但是现在，系统常常提供这两者。而Mac OS X仅支持terminfo。

Goodheart[1991]对terminfo和curses库进行了详细说明，但此书已绝版。Strang[1986]说明了curses函数库的伯克利版本。Strang、Mui和O'Reilly[1988]则对termcap和terminfo进行了说明。

> 可在http://invisible-island.net/ncurses/ncurses.html上找到与SVR4 curses接口兼容的自由版本的ncurses函数库。

不论是termcap还是terminfo，它们本身都不处理本章所述及的问题，如更改终端的模式、更改终端特殊字符以及处理窗口大小等等。它们所提供的是在各种终端上执行一般性操作（清屏、移动光标）的方法。另一方面，在本章所述问题方面，curses能提供某种具体细节方面的帮助。curses提供了很多函数，包括设置原始模式、设置cbreak模式、打开和关闭回显等等。但是curses是为字符型终端设计的，而现在字符型终端大部分已被以像素为基础的图形终端所代替。

672

18.14 小结

终端有很多特征和选项，其中大多数都可按需进行改变。本章说明了很多更改终端操作的函数，比如更改特殊输入字符和可选择标志的函数，还介绍了可对终端设备设置的各个终端特殊字符以及很多选项。

终端的输入模式有规范的（每次一行）和非规范的两种。本章中包含了若干这两种工作模式的实例，也提供了一些函数，它们在POSIX.1终端选项和较早的BSD cbreak及原始模式之间进行变换。本章还说明了如何取得和改变终端的窗口大小。

习题

18.1 编写一个调用tty_raw并且不恢复终端模式就终止的程序。如果你的系统提供reset(1)命令（本书说明的所有四种平台都提供），使用该命令恢复终端模式。

18.2 c_cflag字段的PARODD标志允许我们设置奇偶校验，而BSD中的tip程序也允许奇偶校验位是0或1，它是如何实现的？

18.3 如果你的系统中stty(1)命令输出MIN和TIME值，做下面的练习。两次登录系统，其中一次登录时打开vi编辑器，在另外一次登录中用stty命令确定vi设置的MIN和TIME值(vi将终端设置为非规范模式)。(如果在你的终端上运行窗口系统，那么你也可以进行同样的测试，方法是：登录一次，然后用两个分开的窗口。)

673

第19章

伪 终 端

19.1 引言

在第9章，我们了解到终端登录是经由自动提供终端语义的终端设备进行的。在终端和运行的程序之间有一个终端行规程（见图18-2），通过它我们能够在终端上设置特殊字符（退格、行删除、中断等）。但是，当一个登录请求到达网络连接时，终端行规程并不是自动被加载到网络连接和登录程序shell之间的。图9-5显示了一个用来提供终端语义的*伪终端*（pseudo-terminal）设备驱动程序。

除了用于网络登录，伪终端还被用在其他方面，本章将对此进行介绍。我们首先概要叙述如何使用伪终端，接着讨论某些具体的使用例子。然后，我们提供在不同平台上用于创建伪终端的函数，并使用这些函数编写一个程序，我们将该程序称为pty。我们将看到pty程序的一些不同用途：抄录在终端上输入和输出的所有字符（script(1)程序）；运行协同进程来避免程序清单15-10中遇到的缓冲问题。

19.2 概述

*伪终端*这个术语暗示对于一个应用程序而言，它看上去像一个终端，但事实上伪终端并不是一个真正的终端。图19-1显示了使用伪终端时相关进程的典型结构。其中关键点如下：

675

- 通常一个进程打开伪终端主设备，然后调用fork。子进程建立了一个新的会话，打开一个相应的伪终端从设备，将其文件描述符复制到标准输入、标准输出和标准出错，然后调用exec。伪终端从设备成为子进程的控制终端。
- 对于伪终端从设备之上的用户进程来说，其标准输入、标准输出和标准出错都是终端设备。对于这些文件描述符，用户进程能够调用第18章中说明的所有输入/输出函数。但是因为在伪终端从设备之下并没有真正的终端设备，无意义的函数调用（改变波特率、发送中断符、设置奇偶校验等）将被忽略。
- 任何写到伪终端主设备的东西都会作为从设备的输入，反之亦然。事实上所有从设备端的输入都来自于伪终端主设备上的用户进程。这看起来就像一个双向管道，但从设备上的终端行规程使我们拥有普通管道没有的其他处理能力。

图19-1显示了FreeBSD、Mac OSX或Linux系统中伪终端的结构。19.3.2节和19.3.3节将介绍如何打开这些设备。

在Solaris中，伪终端是使用STREAMS子系统构建的（见14.4节）。图19-2详细描述了Solaris中各个伪终端STREAMS模块之间的关系。虚线框中的两个STREAMS模块是可选的。pckt和ptem模块帮助提供伪终端特有的语义。另外两个模块（ldterm和ttcompat）提供了行规程处理。

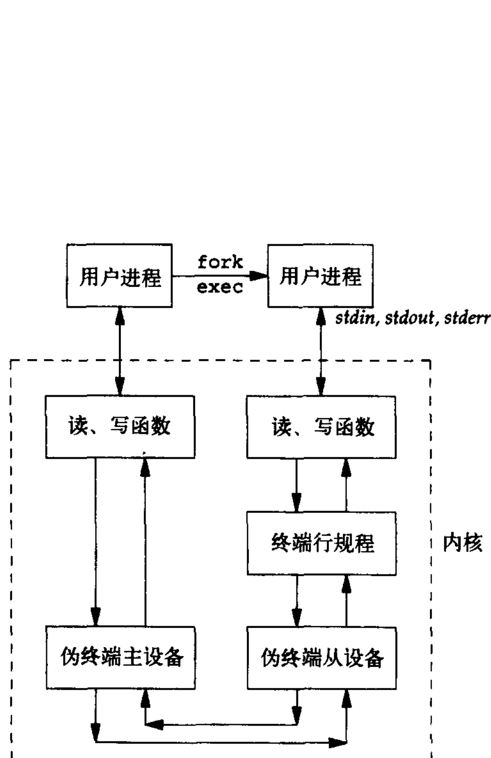


图19-1　使用伪终端的相关进程的典型结构

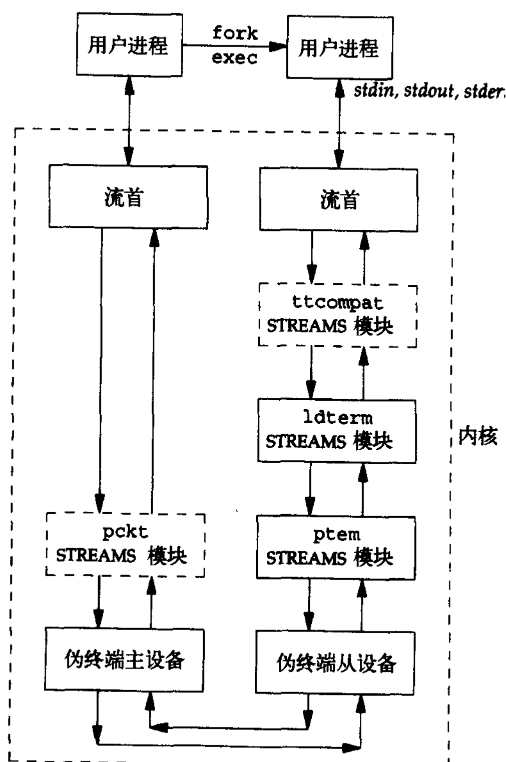


图19-2　Solaris中的伪终端结构

676

请注意从设备上的三个STREAMS模块与程序清单14-9（网络登录）的输出是一样的。19.3.1节将介绍如何组织这些STREAMS模块。

从现在开始将简化以上图示，不再画出图19-1中的“读、写函数”或图19-2中的“流首”。使用缩写“PTY”表示伪终端，并将图19-2中所有伪终端从设备之上的STREAMS模块合并在一起表示为“终端行规程”模块，这与图19-1类似。

现在，我们来观察伪终端的某些典型用途。

1. 网络登录服务器

伪终端可用于构造网络登录服务器。典型的例子是telnetd和rlogind服务器。Stevens[1990]第15章详细讨论了提供rlogin服务的步骤。一旦登录shell运行在远端主机上，即可得到如图19-3的结构。telnetd服务器使用类似的结构。

677

在rlogind服务器和登录shell之间有两个exec调用，这是因为login程序通常是在两个exec之间检验用户是否合法。

本图的一个关键点是驱动PTY主设备的进程通常同时在读写另一个I/O流。图中另一个I/O流是TCP/IP框。这表示该进程必然使用了如select或poll那样的I/O多路转接（见14.5节），或被分成两个进程或线程。

2. script程序

script(1)程序是随大多数UNIX系统提供的，它将终端会话的所有输入和输出信息复制到一个文件中。它将自己置于终端和登录shell的一个新调用之间，从而完成此工作。图19-4详细描述了script程序中有关的交互。这里要特别指出，script程序通常是从登录shell启动的，该shell然后等待script程序的结束。

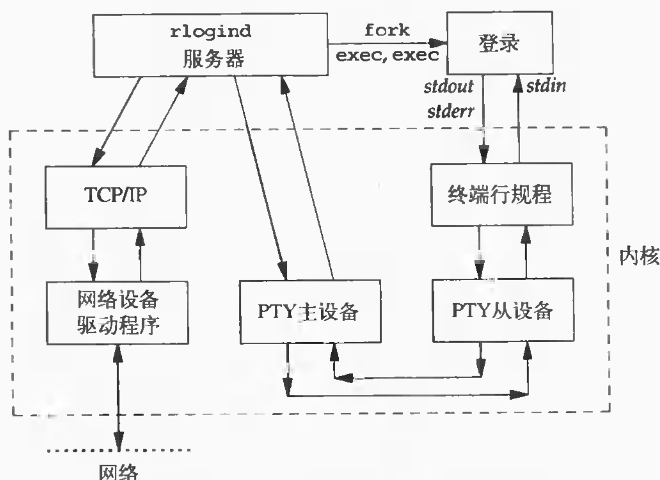


图19-3　rlogind服务器的进程组织结构

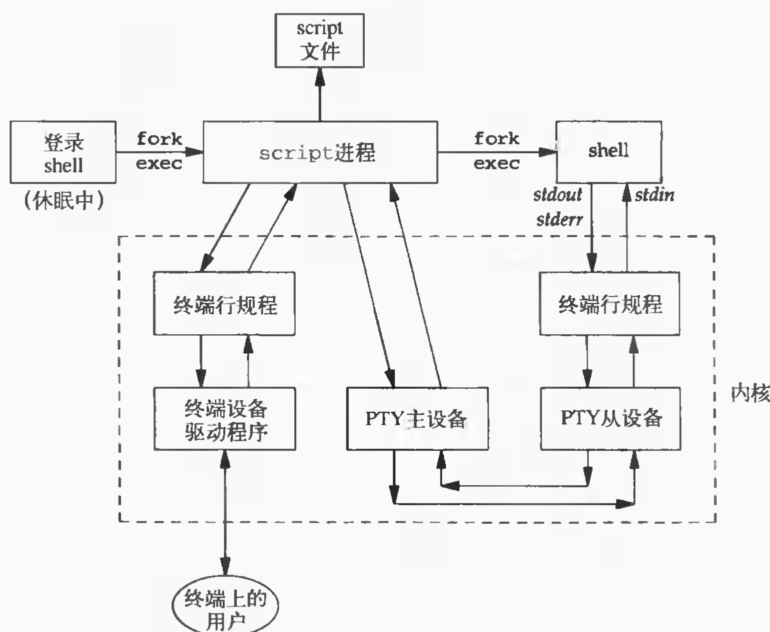


图19-4　script程序

script程序运行时，位于PTY从设备之上的终端行规程的所有输出都被复制到script文件中（通常叫做typescript）。因为击键通常由该行规程模块回显，所以该script文件也包括了输入的内容。但是，因为键入的口令不被回显，所以该文件不会包含口令。

> 在编写本书第1版时，Rich Stevens用script程序获取实例程序的输出。这样避免了手工复制程序输出可能带来的错误。但是，使用script的不足之处是必须处理script文件中的控制字符。

678

在19.5节开发了通用的pty程序后，我们将看到使用pty程序和一个简单的shell脚本就能够实现一种script程序版本。

3. expect程序

伪终端可以用来在非交互模式中驱动交互式程序的运行。许多程序需要一个终端才能运行，

passwd(1)命令就是一个例子，它要求用户在系统提示后输入口令。

为了支持批处理操作模式而修改所有交互式程序，是非常麻烦的。相比之下，一个更好的解决方法是通过一个脚本来驱动交互式程序。expect程序[Libes 1990，1991，1994]提供了这样的方法。类似于19.5节的pty程序，它使用伪终端来运行其他程序。并且，expect还提供了一种编程语言用于检查运行程序的输出，以确定用什么作为输入发送给该程序。当一个源自脚本的交互式的程序正在运行时，不能仅仅是将脚本中的所有内容输入到程序中去，或将程序的输出送至脚本，而是要检查程序的输出，从而决定下一步输入的内容。

679

4. 运行协同进程

在程序清单15-10所示的协同进程例子中，我们不能调用使用标准I/O库进行输入、输出的协同进程，这是因为当通过管道与协同进程进行通信时，标准I/O库会将标准输入和输出的内容放到缓冲区中，从而引起死锁。如果协同进程是一个已经编译的程序而我们又没有源程序，则无法在源程序中加入fflush语句来解决这个问题。图15-8显示了一个进程驱动协同进程的情况。我们须要做的是将一个伪终端放到两个进程之间(见图19-5)，这诱使协同进程认为它是由终端而非另一个进程驱动的。

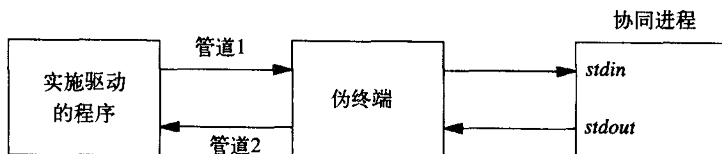


图19-5　用伪终端驱动一个协同进程

现在协同进程的标准输入和标准输出就像终端设备一样，所以标准I/O库会将这两个流设置为行缓冲。

父进程有两种方法在自身和协同进程之间获得伪终端（这种情况下的父进程可以类似程序清单15-9，使用两个管道和协同进程进行通信；或者像程序清单17-1那样，使用一个STREAMS管道）。一种方法是父进程直接调用pty_fork函数（见19.4节）而不是fork。另一种方法是将协同进程作为参数来调用exec执行该pty程序（见19.5节）。我们将在说明pty程序后介绍这两种方法。

5. 观看长时间运行程序的输出

使用任何一个标准shell，可以将一个需要长时间运行的程序放到后台运行。但是如果将该程序的标准输出重定向到一个文件，并且它产生的输出又不多，那么我们就不能方便地监控程序的进展，这是因为标准I/O库会将标准输出全部放在缓冲区中。我们看到的将只是标准I/O库函数写到输出文件中的成块输出，有时甚至可能是大到8192字节的一块。

如果有源程序，则可以加入fflush调用。另一种方法是，可以在pty程序下运行该程序，让标准I/O库认为标准输出是终端。图19-6说明了这个结构，我们将这个缓慢输出的程序称为slowout。从登录shell到pty进程的fort/exec箭头用虚线表示，以强调pty进程是作为后台任务运行的。

680

19.3　打开伪终端设备

各种平台打开伪终端设备的方法有所不同。在Single UNIX Specification的XSI扩展中包含了很多函数，试图统一这些方法。这些函数的基础是SVR4用于管理基于STREAMS的伪终端的一组函数。

posix_openpt用来打开下一个可用的伪终端主设备，该函数是可移植的。

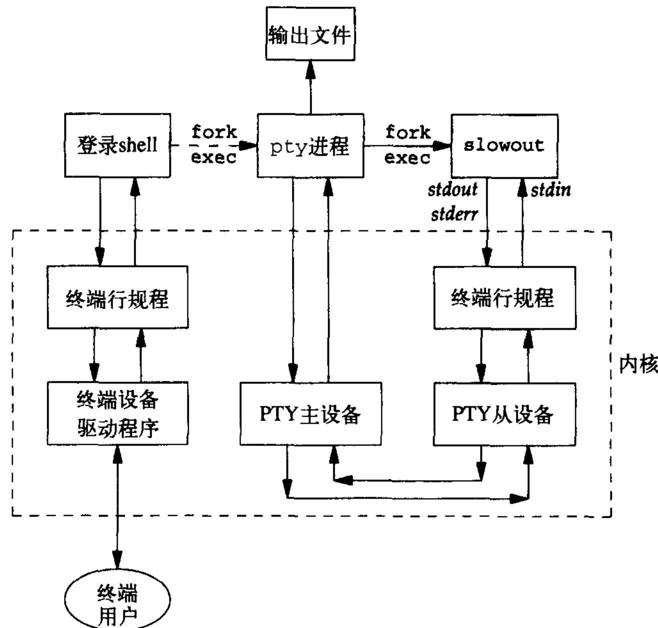


图19-6 使用伪终端运行一个缓慢输出的程序

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

返回值：若成功则返回下一个可用的PTY主设备的文件描述符，若出错则返回-1

参数*oflag*是一个位屏蔽字，指定如何打开主设备，它类似于open(2)的*oflag*参数，但是并不支持所有打开标志。对于posix_openpt，我们可以指定O_RDWR，要求打开主设备进行读、写；可以指定O_NOCTTY以防止主设备成为调用者的控制终端。其他打开标志都会导致未定义的行为。

681

在伪终端从设备可被使用之前，必须设置它的权限，使得应用程序可以访问它。grantpt函数提供这样的功能。它把从设备节点的用户ID设置为调用者的实际用户ID，设置其组ID为一非指定值，通常是可以访问该终端设备的组。将权限设置为：对单个所有者是读/写，对组所有者是写（0620）。

```
#include <stdlib.h>

int grantpt(int filedes);

int unlockpt(int filedes);
```

两个函数的返回值：若成功则返回0，若出错则返回-1

为了更改从设备节点的权限，grantpt可能需要fork以及exec一个设置用户ID程序（例如在Solaris中是/usr/lib/pt_chmod）。于是，如果调用者正在捕捉SIGCHLD信号，那么其运行行为是没有说明的。

unlockpt函数用于准予对伪终端从设备的访问，从而允许应用程序打开该设备。阻止其他进程打开从设备后，建立该设备的应用程序有机会在使用主、从设备之前正确地初始化这些设备。

注意，在grantpt和unlockpt这两个函数中，文件描述符参数是与主伪终端设备关联的文件描述符。

ptsname函数用于在给定主伪终端设备的文件描述符时，找到从伪终端设备的路径名。这使应用程序可以独立于给定平台的某种惯例而标识从设备。注意，该函数返回的名字可能存放在静态存储区中，所以以后的调用可能会覆盖它。

```
#include <stdlib.h>

char *ptsname(int filedes);
```

返回值：若成功则返回指向PTY从设备名的指针，若出错则返回NULL

表19-1摘要列出了Single UNIX Specification中的伪终端函数，指出了本书讨论的四种UNIX系统支持哪些函数。

表19-1　XSI伪终端函数

函　数	说　明	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
grantpt	更改从PTY设备的权限	•	•	•		•
posix_openpt	打开一主PTY设备	•	•			
ptsname	返回从PTY设备的名字	•	•	•		•
unlockpt	允许从PTY设备被打开	•	•	•		•

682

在FreeBSD中，unlockpt不执行任何操作；只是为了调用posix_openpt的应用程序的兼容性而定义了O_NOCTTY标志。在FreeBSD中打开终端设备并不会导致分配控制终端的副作用，所以O_NOCTTY标志并无作用。

尽管Single UNIX Specification已试图改善在此方面的可移植性，但如表19-1所示，实现还是落后于标准，正在逐步赶上。于是，我们提供了ptym_open和ptys_open这两个函数处理所有的细节，前者打开下一个可用的PTY主设备，后者打开相应的从设备。

```
#include "apue.h"

int ptym_open(char *pts_name, int pts_namesz);
```

返回值：若成功则返回PTY主设备的文件描述符，若出错则返回−1

```
int ptys_open(char *pts_name);
```

返回值：若成功则返回PTY从设备的文件描述符，若出错则返回−1

通常我们不直接调用这两个函数，函数pty_fork（见19.4节）调用它们，并且也用fork产生出一个子进程。

ptym_open确定下一个可用的PTY主设备并打开该设备。调用者必须分配一个数组来存放主设备或从设备的名字，并且如果调用成功，相应的从设备名字会通过*pts_name*返回。然后，把这个名字传给ptys_open，ptys_open函数用来打开该从设备。缓冲区的长度由*pts_namesz*传送，所以ptym_open函数不复制比该缓冲区长的字符串。

在说明了pty_fork函数之后，就容易明白为何要提供两个函数来打开这两个设备。通常，一个进程调用ptym_open来打开一个主设备并且得到从设备的名称。该进程然后以fork分出子进程，子进程在调用setsid建立新的会话后调用ptys_open打开从设备。这就是从设备如

何成为子进程的控制终端的过程。

19.3.1 基于STREAMS的伪终端

Solaris下所有伪终端的STREAMS实现细节在Sun Microsystems[2002]的Appendix C中有所说明。我们通过一个STREAMS克隆设备（clone device）对下一个可用的PTY主设备进行访问。克隆设备是一个特殊设备，打开它时返回一个未用设备。（在Rago[1993]中，详细讨论了STREAMS克隆设备的打开。）

基于STREAMS的PTY主克隆设备是`/dev/ptmx`。当我们打开该设备，其克隆open例程自动决定第一个未被使用的PTY主设备，并打开这个设备（见程序清单19-1）。（下一节将看到在基于BSD的系统中，我们必须自己找到第一个未被使用的PTY主设备。）

程序清单19-1 基于STREAMS的伪终端打开函数

683

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <stropts.h>

int
ptym_open(char *pts_name, int pts_namesz)
{
    char    *ptr;
    int     fdm;

    /*
     * Return the name of the master device so that on failure
     * the caller can print an error message.  Null terminate
     * to handle case where strlen("/dev/ptmx") > pts_namesz.
     */
    strncpy(pts_name, "/dev/ptmx", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    if ((fdm = open(pts_name, O_RDWR)) < 0)
        return(-1);
    if (grantpt(fdm) < 0) {     /* grant access to slave */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) {    /* clear slave's lock flag */
        close(fdm);
        return(-3);
    }
    if ((ptr = ptsname(fdm)) == NULL) { /* get slave's name */
        close(fdm);
        return(-4);
    }

    /*
     * Return name of slave.  Null terminate to handle
     * case where strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);            /* return fd of master */
}

int
```

```
ptys_open(char *pts_name)
{
    int     fds, setup;

    /*
     * The following open should allocate a controlling terminal.
     */
    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);

    /*
     * Check if stream is already set up by autopush facility.
     */
    if ((setup = ioctl(fds, I_FIND, "ldterm")) < 0) {
        close(fds);
        return(-6);
    }
    if (setup == 0) {
        if (ioctl(fds, I_PUSH, "ptem") < 0) {
            close(fds);
            return(-7);
        }
        if (ioctl(fds, I_PUSH, "ldterm") < 0) {
            close(fds);
            return(-8);
        }
        if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
            close(fds);
            return(-9);
        }
    }
    return(fds);
}
```

684

首先用open打开克隆设备/dev/ptmx，得到PTY主设备的文件描述符。打开这个主设备自动锁定了对应的从设备。

然后调用grantpt来改变从设备的访问权限。在Solaris中，它执行如下操作：将从设备的所有权改为实际用户ID；将组所有权改为组tty；将权限改为只允许用户-读，用户-写和组-写。将组所有权设置为tty并允许组-写权限是因为程序wall(1)和write(1)是设置-组-ID的，并且组为tty。调用函数grantpt执行程序/usr/lib/pt_chmod。该程序是设置-用户-ID的，并且用户为root，因此它能够修改从设备的所有者和权限。

函数unlockpt用来清除从设备的内部锁。在打开从设备前必须做这件事情。另外，我们必须调用ptsname来得到从设备的名字。这个名字的格式是/dev/pts/*NNN*。

接下来的函数是ptys_open，该函数被用来真正打开一个从设备。Solaris沿袭了系统V的处理模式，如果调用者是一个还没有控制终端的会话首进程，调用open就会分配一个PTY从设备作为它的控制终端。如果不希望该函数自动做这件事，可以在调用open时指明O_NOCTTY标志。

打开从设备后，将三个STREAMS模块压入从设备的流中。ptem是伪终端仿真模块，ldterm是终端行规程模块，这两个模块合在一起像一个真正的终端一样工作。ttcompat提供了向早期系统（如V7、4BSD和Xenix）的ioctl调用的兼容性。这是一个可选的模块，但是因为对于控制台登录和网络登录，它是自动被压入的（见程序清单14-9程序的输出），所以我们将其压入到从设备的流中。

我们可能并不需要压入这三个模块，其原因是，它们可能已经位于流中。STREAMS系统支

持称为autopush（自动压入）的设施，它帮助系统管理员配置一张模块列表，只要打开一特定设备，就将这些模块压入流中（详见Rago[1993]）。我们使用I_FIND ioctl命令观察ldterm是否已在流中。如果是，则认为该流已用autopush机制配置，这样就无需再重复压入相应模块。 685

调用ptym_open和ptys_open函数的结果是在调用进程中打开了两个文件描述符，一个是针对主设备的，另一个是针对从设备的。

19.3.2 基于BSD的伪终端

在基于BSD和基于Linux的系统中，我们提供了自己的XSI函数版本，根据基础平台提供了哪些函数，我们可选地将它们包括在我们的函数库中。

在我们的posix_openpt版本（见程序清单19-2）中，我们必须自己确定第一个可用的PTY主设备。为达到此目的，从/dev/ptyp0开始并不断尝试，直到成功打开一个可用的PTY主设备或试完了所有设备。在打开设备时，可能得到以下两种不同的错误：EIO指示设备已经被使用；ENOENT表示设备不存在。当检测到后一种情况，这表明所有的PTY设备都已在使用之中，于是就可以停止搜索。一旦成功地打开一个例如名为/dev/pty*MN*的PTY主设备，那么对应的从设备名为/dev/tty*MN*。在Linux中，如果PTY主设备名是/dev/pty/m*XX*，那么对应的PTY从设备名是/dev/pty/s*XX*。

程序清单19-2 BSD和Linux的伪终端打开函数

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <grp.h>

#ifndef _HAS_OPENPT
int
posix_openpt(int oflag)
{
    int     fdm;
    char    *ptr1, *ptr2;
    char    ptm_name[16];

    strcpy(ptm_name, "/dev/ptyXY");
    /* array index:   0123456789   (for references in following code) */
    for (ptr1 = "pqrstuvwxyzPQRST"; *ptr1 != 0; ptr1++) {
        ptm_name[8] = *ptr1;
        for (ptr2 = "0123456789abcdef"; *ptr2 != 0; ptr2++) {
            ptm_name[9] = *ptr2;

            /*
             * Try to open the master.
             */
            if ((fdm = open(ptm_name, oflag)) < 0) {
                if (errno == ENOENT)    /* different from EIO */
                    return(-1);         /* out of pty devices */
                else
                    continue;           /* try next pty device */
            }
            return(fdm);        /* got it, return fd of master */
        }
    }
    errno = EAGAIN;
    return(-1);     /* out of pty devices */
```

686

```
}
#endif

#ifndef _HAS_PTSNAME
char *
ptsname(int fdm)
{
    static char pts_name[16];
    char        *ptm_name;

    ptm_name = ttyname(fdm);
    if (ptm_name == NULL)
        return(NULL);
    strncpy(pts_name, ptm_name, sizeof(pts_name));
    pts_name[sizeof(pts_name) - 1] = '\0';
    if (strncmp(pts_name, "/dev/pty/", 9) == 0)
        pts_name[9] = 's';  /* change /dev/pty/mXX to /dev/pty/sXX */
    else
        pts_name[5] = 't';  /* change "pty" to "tty" */
    return(pts_name);
}
#endif

#ifndef _HAS_GRANTPT
int
grantpt(int fdm)
{
    struct group    *grptr;
    int             gid;
    char            *pts_name;

    pts_name = ptsname(fdm);
    if ((grptr = getgrnam("tty")) != NULL)
        gid = grptr->gr_gid;
    else
        gid = -1;       /* group tty is not in the group file */

    /*
     * The following two calls won't work unless we're the superuser.
     */
    if (chown(pts_name, getuid(), gid) < 0)
        return(-1);
    return(chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP));
}
#endif

#ifndef _HAS_UNLOCKPT
int
unlockpt(int fdm)
{
    return(0); /* nothing to do */
}
#endif

int
ptym_open(char *pts_name, int pts_namesz)
{
    char    *ptr;
    int     fdm;

    /*
     * Return the name of the master device so that on failure
     * the caller can print an error message.  Null terminate
```

687

```
	 * to handle case where string length > pts_namesz.
	 */
	strncpy(pts_name, "/dev/ptyXX", pts_namesz);
	pts_name[pts_namesz - 1] = '\0';
	if ((fdm = posix_openpt(O_RDWR)) < 0)
		return(-1);
	if (grantpt(fdm) < 0) {		/* grant access to slave */
		close(fdm);
		return(-2);
	}
	if (unlockpt(fdm) < 0) {	/* clear slave's lock flag */
		close(fdm);
		return(-3);
	}
	if ((ptr = ptsname(fdm)) == NULL) {	/* get slave's name */
		close(fdm);
		return(-4);
	}

	/*
	 * Return name of slave.  Null terminate to handle
	 * case where strlen(ptr) > pts_namesz.
	 */
	strncpy(pts_name, ptr, pts_namesz);
	pts_name[pts_namesz - 1] = '\0';
	return(fdm);			/* return fd of master */
}

int
ptys_open(char *pts_name)
{
	int fds;

	if ((fds = open(pts_name, O_RDWR)) < 0)
		return(-5);
	return(fds);
}
```

688

在我们的grantpt函数版本中，调用了chown和chmod函数，但是必须意识到调用这两个函数的进程必须有超级用户权限，否则这两个函数都不能正常工作。如果必须改变所有权或权限保护，那么这两个函数调用必须放在一个设置用户ID的root用户的可执行程序中，这类似于Solaris实现grantpt函数的方法。

程序清单19-2中的函数ptys_open简单地打开该从设备，不需要其他初始化操作。在基于BSD的系统之下打开PTY从设备不具有分配该设备作为控制终端的副作用。19.4节将探讨如何在基于BSD的系统下分配控制终端。

> 我们的posix_openpt函数版本尝试16组不同的PTY主设备：从/dev/ptyp0到/dev/ptyTf。可用的实际PTY设备数取决于两个因素：(a) 内核中配置的数量；(b) 在/dev目录中已创建的特殊设备文件数。对于任何程序来说，可用数量是(a)和(b)中较小的那一个。

19.3.3 基于Linux的伪终端

Linux支持访问伪终端的BSD方法，所以程序清单19-2中所示的函数对于Linux同样适用。然而，Linux也支持使用/dev/ptmx（但这不是STREAMS设备）的克隆风格的伪终端接口。该克隆风格接口要求额外步骤去标识和解锁从设备。在Linux里，我们可以使用程序清单19-3中

的函数访问从设备。

程序清单19-3 Linux的伪终端open函数

```
#include "apue.h"
#include <fcntl.h>

#ifndef _HAS_OPENPT
int
posix_openpt(int oflag)
{
    int     fdm;

    fdm = open("/dev/ptmx", oflag);
    return(fdm);
}
#endif

#ifndef _HAS_PTSNAME
char *
ptsname(int fdm)
{
    int         sminor;
    static char pts_name[16];

    if (ioctl(fdm, TIOCGPTN, &sminor) < 0)
        return(NULL);
    snprintf(pts_name, sizeof(pts_name), "/dev/pts/%d", sminor);
    return(pts_name);
}
#endif

#ifndef _HAS_GRANTPT
int
grantpt(int fdm)
{
    char            *pts_name;

    pts_name = ptsname(fdm);
    return(chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP));
}
#endif

#ifndef _HAS_UNLOCKPT
int
unlockpt(int fdm)
{
    int lock = 0;

    return(ioctl(fdm, TIOCSPTLCK, &lock));
}
#endif

int
ptym_open(char *pts_name, int pts_namesz)
{
    char    *ptr;
    int     fdm;

    /*
     * Return the name of the master device so that on failure
     * the caller can print an error message.  Null terminate
     * to handle case where string length > pts_namesz.
```

689

```
     */
    strncpy(pts_name, "/dev/ptmx", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';

    fdm = posix_openpt(O_RDWR);
    if (fdm < 0)
        return(-1);
    if (grantpt(fdm) < 0) {     /* grant access to slave */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) {    /* clear slave's lock flag */
        close(fdm);
        return(-3);
    }
    if ((ptr = ptsname(fdm)) == NULL) { /* get slave's name */
        close(fdm);
        return(-4);
    }

    /*
     * Return name of slave.  Null terminate to handle case
     * where strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);            /* return fd of master */
}

int
ptys_open(char *pts_name)
{
    int fds;

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);
    return(fds);
}
```

690

在Linux中，PTY从设备已为组tty所拥有，所以在grantpt中须做的一切是确保访问权限是正确的。

19.4 pty_fork函数

现在使用上一节介绍的两个函数ptym_open和ptys_open，编写我们称之为pty_fork的函数。这个新函数具有如下功能：用fork调用打开主设备和从设备，创建作为会话首进程的子进程并使其具有控制终端。

```
#include "apue.h"
#include <termios.h>
#include <sys/ioctl.h>  /* find struct winsize on BSD systems */

pid_t pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
               const struct termios *slave_termios,
               const struct winsize *slave_winsize);
```

返回值：子进程中返回0，父进程中返回子进程的进程ID，出错返回-1

PTY主设备的文件描述符通过*ptrfdm*指针返回。

如果*slave_name*不为空，从设备名就被存放在该指针指向的存储区中。调用者必须为该存储区分配空间。

如果指针*slave_termios*不为空，则使用该指针所引用的结构，初始化从设备的终端行规程。如果该指针为空，那么，系统把从设备的termios结构设置为实现定义的初始状态。类似地，如果*slave_winsize*指针不为空，那么按该指针所引用的结构初始化从设备的窗口大小。如果该

691

指针为空，winsize结构通常被初始化为0。

程序清单19-4显示了该函数的代码。它调用相应的ptym_open和ptys_open函数，在本书讨论的四种平台上都能适用。

程序清单19-4　pty_fork函数

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

pid_t
pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
         const struct termios *slave_termios,
         const struct winsize *slave_winsize)
{
    int     fdm, fds;
    pid_t   pid;
    char    pts_name[20];

    if ((fdm = ptym_open(pts_name, sizeof(pts_name))) < 0)
        err_sys("can't open master pty: %s, error %d", pts_name, fdm);

    if (slave_name != NULL) {
        /*
         * Return name of slave.  Null terminate to handle case
         * where strlen(pts_name) > slave_namesz.
         */
        strncpy(slave_name, pts_name, slave_namesz);
        slave_name[slave_namesz - 1] = '\0';
    }

    if ((pid = fork()) < 0) {
        return(-1);
    } else if (pid == 0) {      /* child */
        if (setsid() < 0)
            err_sys("setsid error");

        /*
         * System V acquires controlling terminal on open().
         */
        if ((fds = ptys_open(pts_name)) < 0)
            err_sys("can't open slave pty");
        close(fdm);     /* all done with master in child */

#if defined(TIOCSCTTY)
        /*
         * TIOCSCTTY is the BSD way to acquire a controlling terminal.
         */
        if (ioctl(fds, TIOCSCTTY, (char *)0) < 0)
            err_sys("TIOCSCTTY error");
#endif
```

692

```
        /*
         * Set slave's termios and window size.
         */
        if (slave_termios != NULL) {
            if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                err_sys("tcsetattr error on slave pty");
        }
        if (slave_winsize != NULL) {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
                err_sys("TIOCSWINSZ error on slave pty");
        }

        /*
         * Slave becomes stdin/stdout/stderr of child.
         */
        if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
            err_sys("dup2 error to stderr");
        if (fds != STDIN_FILENO && fds != STDOUT_FILENO &&
          fds != STDERR_FILENO)
            close(fds);
        return(0);      /* child returns 0 just like fork() */
    } else {                    /* parent */
        *ptrfdm = fdm;  /* return fd of master */
        return(pid);    /* parent returns pid of child */
    }
}
```

693

在打开PTY主设备后，调用fork。正如前面提到的，子进程先调用setsid建立新的会话，然后才调用ptys_open。当调用setsid时，子进程还不是一个进程组的首进程，因此执行9.5节中列出的三个操作步骤：(a) 为子进程创建一个新的会话，它是该会话首进程；(b) 为子进程创建一个新的进程组；(c) 子进程断开与以前可能有的控制终端的关联，于是不再有控制终端。在Linux和Solaris系统中，当调用ptys_open时，从设备成为新会话的控制终端。在FreeBSD和Mac OS X系统中，必须调用ioctl并使用参数TIOCSCTTY来分配一个控制终端（Linux也支持TIOCSCTTY ioctl命令）。然后termios和winsize这两个结构在子进程中被初始化。最后从设备的文件描述符被复制到子进程的标准输入、标准输出和标准出错中。这意味着不管子进程以后调用exec执行何种进程，它都具有同PTY从设备（其控制终端）联系起来的上述三个描述符。

在调用fork后，父进程返回PTY主设备的描述符以及子进程ID。下一节将在pty程序中使用pty_fork函数。

19.5 pty程序

编写pty程序的目的是为了用键入：

```
pty prog arg1 arg2
```

来代替：

```
prog arg1 arg2
```

当我们用pty来执行另一个程序时，该程序在一个它自己的会话中执行，并和一个伪终端

连接。

让我们查看pty程序的源代码。程序清单19-5包含main函数。它调用上一节的pty_fork函数。

程序清单19-5 pty程序的main函数

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>  /* for struct winsize */
#endif

#ifdef LINUX
#define OPTSTR "+d:einv"
#else
#define OPTSTR "d:einv"
#endif

static void set_noecho(int);    /* at the end of this file */
void        do_driver(char *);  /* in the file driver.c */
void        loop(int, int);     /* in the file loop.c */

int
main(int argc, char *argv[])
{
    int             fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t           pid;
    char            *driver;
    char            slave_name[20];
    struct termios  orig_termios;
    struct winsize  size;

    interactive = isatty(STDIN_FILENO);
    ignoreeof = 0;
    noecho = 0;
    verbose = 0;
    driver = NULL;

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ((c = getopt(argc, argv, OPTSTR)) != EOF) {
        switch (c) {
        case 'd':       /* driver for stdin/stdout */
            driver = optarg;
            break;
        case 'e':       /* noecho for slave pty's line discipline */
            noecho = 1;
            break;

        case 'i':       /* ignore EOF on standard input */
            ignoreeof = 1;
            break;

        case 'n':       /* not interactive */
            interactive = 0;
            break;

        case 'v':       /* verbose */
            verbose = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
```

694

```
        }
    }
    if (optind >= argc)
        err_quit("usage: pty [ -d driver -einv ] program [ arg ... ]");

    if (interactive) {  /* fetch current termios and window size */
        if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
            err_sys("tcgetattr error on stdin");
        if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
            err_sys("TIOCGWINSZ error");
        pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
          &orig_termios, &size);
    } else {
        pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
          NULL, NULL);
    }

    if (pid < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        if (noecho)
            set_noecho(STDIN_FILENO);   /* stdin is slave pty */

        if (execvp(argv[optind], &argv[optind]) < 0)
            err_sys("can't execute: %s", argv[optind]);
    }

    if (verbose) {
        fprintf(stderr, "slave name = %s\n", slave_name);
        if (driver != NULL)
            fprintf(stderr, "driver = %s\n", driver);
    }

    if (interactive && driver == NULL) {
        if (tty_raw(STDIN_FILENO) < 0)  /* user's tty to raw mode */
            err_sys("tty_raw error");
        if (atexit(tty_atexit) < 0)     /* reset user's tty on exit */
            err_sys("atexit error");
    }

    if (driver)
        do_driver(driver);  /* changes our stdin/stdout */

    loop(fdm, ignoreeof);   /* copies stdin -> ptym, ptym -> stdout */

    exit(0);
}

static void
set_noecho(int fd)      /* turn off echo (for slave pty) */
{
    struct termios  stermios;

    if (tcgetattr(fd, &stermios) < 0)
        err_sys("tcgetattr error");

    stermios.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);

    /*
     * Also turn off NL to CR/NL mapping on output.
     */
    stermios.c_oflag &= ~(ONLCR);

    if (tcsetattr(fd, TCSANOW, &stermios) < 0)
```

695

```
        err_sys("tcsetattr error");
}
```

下一节介绍pty程序的不同用法时，将看到多种命令行的选项。getopt函数帮助我们以协调一致的模式分析命令行参数。我们将在第21章更详细地讨论该函数。

在调用pty_fork前，我们取termios和winsize结构的当前值，将其作为参数传递给pty_fork。通过这种方法，PTY从设备具有和当前终端相同的初始状态。

从pty_fork返回后，子进程可选择地关闭了PTY从设备的回送，并调用execvp来执行命令行指定的程序。所有余下的命令行参数将成为该程序的参数。

父进程可选地将用户终端设置为原始模式。在这种情况下，父进程也设置退出处理程序，使得在调用exit时复原终端状态，下一节将讨论do_driver函数。

接下来，父进程调用函数loop（见程序清单19-6）。该函数仅仅是将从标准输入接收到的所有内容复制到PTY主设备，并将PTY主设备接收到的所有内容复制到标准输出。尽管使用select或poll的单进程或多线程是可行的，但是为了有所变化，我们程序里使用了两个进程。

696

程序清单19-6　loop函数

```
#include "apue.h"

#define BUFFSIZE    512

static void sig_term(int);
static volatile sig_atomic_t    sigcaught; /* set by signal handler */

void
loop(int ptym, int ignoreeof)
{
    pid_t   child;
    int     nread;
    char    buf[BUFFSIZE];

    if ((child = fork()) < 0) {
        err_sys("fork error");
    } else if (child == 0) {     /* child copies stdin to ptym */
        for ( ; ; ) {
            if ((nread = read(STDIN_FILENO, buf, BUFFSIZE)) < 0)
                err_sys("read error from stdin");
            else if (nread == 0)
                break;      /* EOF on stdin means we're done */
            if (writen(ptym, buf, nread) != nread)
                err_sys("writen error to master pty");
        }

        /*
         * We always terminate when we encounter an EOF on stdin,
         * but we notify the parent only if ignoreeof is 0.
         */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM);   /* notify parent */
        exit(0);    /* and terminate; child can't return */
    }

    /*
     * Parent copies ptym to stdout.
     */
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
```

```
        err_sys("signal_intr error for SIGTERM");

    for ( ; ; ) {
        if ((nread = read(ptym, buf, BUFFSIZE)) <= 0)
            break;      /* signal caught, error, or EOF */
        if (writen(STDOUT_FILENO, buf, nread) != nread)
            err_sys("writen error to stdout");
    }

    /*
     * There are three ways to get here: sig_term() below caught the
     * SIGTERM from the child, we read an EOF on the pty master (which
     * means we have to signal the child to stop), or an error.
     */
    if (sigcaught == 0) /* tell child if it didn't send us the signal */
        kill(child, SIGTERM);
    /*
     * Parent returns to caller.
     */
}

/*
 * The child sends us SIGTERM when it gets EOF on the pty slave or
 * when read() fails.  We probably interrupted the read() of ptym.
 */
static void
sig_term(int signo)
{
    sigcaught = 1;      /* just set flag and return */
}
```

697

注意，当使用两个进程时，如果一个终止，那么它必须通知另一个。我们用SIGTERM信号进行这种通知。

19.6 使用pty程序

接下来看几个pty程序的应用实例，并了解使用不同命令行选项的必要性。

如果使用Korn shell，那么我们执行：

```
pty ksh
```

结果得到一个运行在一个伪终端下的全新shell。

如果文件ttyname包含了程序清单18-7中所示的程序，那么可按如下模式执行pty程序：

```
$ who
sar   :0      Oct  5 18:07
sar   pts/0   Oct  5 18:07
sar   pts/1   Oct  5 18:07
sar   pts/2   Oct  5 18:07
sar   pts/3   Oct  5 18:07
sar   pts/4   Oct  5 18:07         pts/4是正在使用的最高PTY设备
$ pty ttyname                      在pty上运行程序清单18-7
fd 0: /dev/pts/5                   pts/5是下一个可用的PTY
fd 1: /dev/pts/5
fd 2: /dev/pts/5
```

1. utmp文件

698

6.8节讨论了记录当前UNIX系统登录用户的utmp文件。那么在伪终端上运行程序的用户是否被认为是登录了呢？如果是远程登录，telnetd和rlogind，显然在伪终端上登录的用户

应该在utmp中有相应记录项。但是，在来自窗口系统或script之类的程序的伪终端上运行shell的用户，是否应该在utmp中有相应记录项呢？对于这个问题一直没有统一的认识。有的系统有记录，有的没有。如果在utmp文件中没有记录的话，who(1)程序一般不会显示相应伪终端正在被使用。

除非utmp允许其他用户的写权限（这被认为是一个安全漏洞），否则使用伪终端的一般程序将不能对其进行写操作。

2. 作业控制交互

在pty下运行作业控制shell时，它能够正常地运行。例如，

```
pty ksh
```

在pty下运行Korn shell。我们能够在这个新shell下运行程序并使用作业控制，这如同在登录shell中一样。但如果在pty下运行一个交互式程序而不是作业控制shell，比如：

```
pty cat
```

在键入作业控制挂起字符之前该程序的运行一切正常。而在键入作业控制挂起字符时，作业控制挂起字符将会被显示为^Z，并且被忽略。在早期基于BSD的系统中，cat进程终止，pty进程终止，回到初始登录shell。为了明白其中的原因，我们需要检查所有相关的进程及其所属的进程组和会话。图19-7显示了pty cat运行时的结构图。

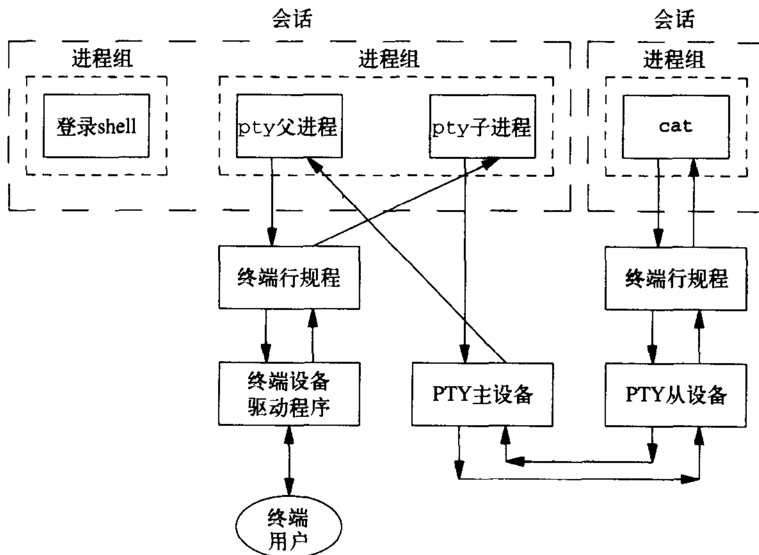


图19-7 pty cat的进程组和会话

键入挂起字符（Ctrl+Z）时，它被cat进程下的行规程模块所识别，这是因为pty将终端（在pty父进程之下）设置为原始模式。但内核不会停止cat进程，这是因为它属于一个孤儿进程组（见9.10节）。cat的父进程是pty父进程，它属于另一个会话。

历史上，不同的系统处理这种情况的方法也不同。POSIX.1只是说明SIGTSTP信号不能被发送给进程。而4.3BSD派生出的系统向进程递送一个它不能捕获的SIGKILL信号。4.4BSD改变了这种做法，转而采用遵循POSIX.1的处理方法。如果SIGTSTP信号具有默认配置，并且传递给孤儿进程组中的一个进程，那么内核无声息地丢弃SIGTSTP信号。大多数当前的实现都采用这种处理模式。

当使用pty来运行作业控制shell时，被这个新shell调用的作业决不会是任何孤儿进程组的成员，这是因为作业控制shell总是属于同一个会话。在这种情况下，键入的Ctrl+Z被发送到由shell调用的进程，而不是shell本身。

让被pty调用的进程能够处理作业控制信号的唯一的方法是，另外增加一个pty命令行标志，使pty子进程自己能够识别作业挂起字符（在pty子进程中），而不是让该字符穿越所有路程而送至另一个行规程模块。 [699]

3. 检查长时间运行程序的输出

另一个使用pty进行作业控制交互的例子见图19-6。如果运行一个缓慢地产生输出的程序：

```
pty slowout > file.out &
```

当子进程试图从标准输入（终端）读入数据时，pty进程立刻停止运行。这是因为该作业是一个后台作业，并且当它试图访问终端时会使作业控制停止。如果将标准输入重定向使得pty不从终端读取数据，如：

```
pty slowout < /dev/null > file.out &
```

那么pty程序也立即停止，这是因为它从标准输入读取到一个文件结束符。解决这个问题的方法是使用-i选项，其含义是忽略来自标准输入的文件结束符：

```
pty -i slowout < /dev/null > file.out &
```

这个标志导致在遇到文件结束符时，程序清单19-6的pty子进程终止，但子进程不会告诉父进程也终止。相反，父进程一直在将PTY从设备的输出复制到标准输出（本例中的file.out）。 [700]

4. script程序

利用pty程序，可以用下面的shell脚本实现script(1)程序：

```
#!/bin/sh
pty "${SHELL:-/bin/sh}" | tee typescript
```

一旦执行这个shell脚本，即可运行ps命令来观察进程之间的关系。图19-8详细地显示了这些关系。

图19-8 script shell脚本

在这个例子中，假设SHELL变量是Korn shell（可能是/bin/ksh）。如前面所述，script仅仅是将新的shell（和它调用的所有的子进程）的输出复制出来，但是因为PTY从设备之上的行规

程模块通常允许回送，所以绝大多数键入都被写到typescript文件中。

5. 运行协同进程

在图15-4中，协同进程不能使用标准I/O函数，其原因是标准输入和输出不指向终端，其输入和输出将被放到缓冲区中。如果用

```
if (execl("./pty", "pty", "-e", "add2", (char *)0) < 0)
```

替代

```
if (execl("./add2", "add2", (char *)0) < 0)
```

在pty下运行协同进程，即使协同进程使用了标准I/O，该程序仍然可以正确运行。

图19-9显示了在使用伪终端作为协同进程的输入和输出时，进程之间的关系。这是图19-5的扩充，它显示了所有的进程间联系和数据流。框中的“实施驱动的程序”是按前面的说明更改了execl的图15-4。

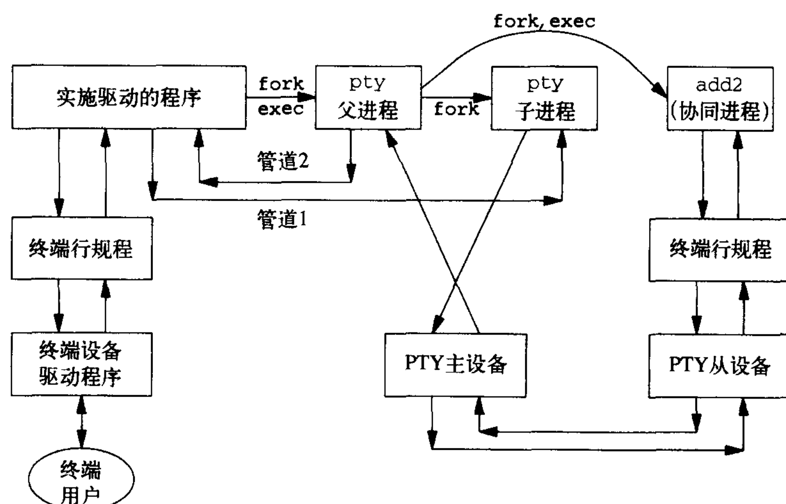


图19-9 运行一协同进程，以伪终端作为其输入和输出

这一实例显示了对于pty程序，-e（不回送）选项的重要性。因为pty的标准输入不连接到终端，所以它不以交互方式运行。在程序清单19-5中，interactive标志默认为false，这是因为对isatty调用的返回是false。这意味着在真正终端之上的行规程保持在规范模式下，并允许回送。指定-e选项后，关掉了PTY从设备上的行规程模块的回送。如果不这样做，则键入的每一个字符都将被两个行规程模块回送两次。

我们也用-e选项关闭termios结构的ONLCR标志，以防止所有的协同进程的输出被回车和换行符终止。

在不同的系统上测试这个例子，会遇到14.8节中描述readn和writen函数时顺便提到的同样问题。当描述符不是引用普通磁盘文件，从read返回时，读到的数据量可能因实现的不同而有所区别。使用pty的协同进程产生了非预期的结果，其原因可追溯至图15-4程序中读管道的read函数，它返回的结果不到一行。解决方法是不使用图15-4的程序，而是使用修改过的使用标准I/O库的习题15.5的程序版本，它将两个管道的标准I/O流都设置为行缓冲。这样，fgets函数将会一直读完一个整行。图15-4中的while循环假设送到协同进程的每一行都会带来一行的返回结果。

6. 用非交互模式驱动交互式程序

虽然让pty运行任一协同进程的想法是非常诱人的，但即使协同进程只是交互式的，它都不能正常工作。问题在于pty只是将其标准输入中的任何内容复制到PTY，并将来自PTY的任何内容复制到标准输出，而并不关心具体发送或得到什么数据。

举个例子，我们可以在pty之下运行telnet命令，直接与远程主机会话：

```
pty telnet 192.168.1.3
```

这样做与直接键入telnet 192.168.1.3相比，并没有带来什么好处，但我们可能希望在一个脚本中运行telnet程序，其目的很可能是要检验远程主机的某些条件。如果telnet.cmd文件包括下面4行：

```
sar
passwd
uptime
exit
```

第1行是登录到远程主机时使用的用户名，第2行是口令字，第3行是希望运行的命令，第4行终止此会话。如果按下列方式运行此脚本

```
pty -i < telnet.cmd telnet 192.168.1.3
```

那么，它不像我们所想的那样操作。所发生的是telnet.cmd文件的内容还没有等到机会提示我们输入账户名和口令之前，就被送到了远程主机。当它关闭回显而读口令时，login使用tcsetattr选项，于是丢弃了已在队列中的所有数据。这样一来，我们发送的数据就被丢掉了。

当以交互方式运行telnet程序时，我们等待远程主机发出输入口令的提示，然后再键入口令，但是pty程序不知道这样做。这就是为什么我们需要一个比pty更巧妙的程序，如expect，来从脚本文件驱动交互式程序。

即使如前所示那样从图15-4运行pty，这也没有任何帮助。这是因为图15-4的程序认为它在一个管道写入的每一行都会在另一个管道产生一行。对于一个交互式程序，输入一行可能产生多行输出。更进一步，图15-4的程序在从协同进程读之前，总是先送一行给该进程。在送给协同进程一些数据之前，我们不可能从协同进程处读。 703

有一些从shell脚本驱动交互式程序的方法。可以在pty上增加一种命令语言和一个解释器。但是一个适当的命令语言可能十倍于pty程序的大小。另一种方案是使用命令语言并用pty_fork函数来调用交互式程序，这正是expect程序所做的。

我们将采用一种不同的途径，使用选项-d使pty程序的输入和输出与驱动进程连接起来。该驱动进程的标准输出是pty的标准输入，反之亦然。这有点儿像协同进程，只是在pty的"另一边"。此种进程结构与图19-9中所示的几乎相同，但是在这种场景中，由pty来完成驱动进程的fork和exec。而且我们在pty和驱动进程之间使用一个双向的流管道，而不是两个半双工管道。

程序清单19-7是do_driver函数的源码，在使用-d选项时，该函数被pty（见程序清单19-5）的main函数调用。

程序清单19-7 pty程序的do_driver函数

```
#include "apue.h"

void
do_driver(char *driver)
```

```
{
    pid_t   child;
    int     pipe[2];

    /*
     * Create a stream pipe to communicate with the driver.
     */
    if (s_pipe(pipe) < 0)
        err_sys("can't create stream pipe");

    if ((child = fork()) < 0) {
        err_sys("fork error");
    } else if (child == 0) {        /* child */
        close(pipe[1]);

        /* stdin for driver */
        if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");

        /* stdout for driver */
        if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (pipe[0] != STDIN_FILENO && pipe[0] != STDOUT_FILENO)
            close(pipe[0]);

        /* leave stderr for driver alone */
        execlp(driver, driver, (char *)0);
        err_sys("execlp error for: %s", driver);
    }
    close(pipe[0]);     /* parent */
    if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
        err_sys("dup2 error to stdin");
    if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("dup2 error to stdout");
    if (pipe[1] != STDIN_FILENO && pipe[1] != STDOUT_FILENO)
        close(pipe[1]);

    /*
     * Parent returns, but with stdin and stdout connected
     * to the driver.
     */
}
```

704

通过我们自己编写由pty调用的驱动程序，可以按所希望的方式驱动交互式程序。即使驱动程序有和pty连接在一起的标准输入和标准输出，它仍然可以通过读、写/dev/tty同用户交互。这个解决方法仍不如expect程序通用，但是它用不到50行的代码，提供了一种可选的方案。

19.7 高级特性

伪终端还有其他特性，我们在这里简略提一下。Sun Microsystems[2002] 和BSD pty(4)的手册页对此有更详细的说明。

1. 打包模式

打包模式（packet mode）能够使PTY主设备了解到PTY从设备的状态变化。在Solaris系统中，可以将流模块pckt压入PTY主设备端来设置这种模式。图19-2显示了这种可选模块。在FreeBSD、Linux和Mac OS X中，可以用TIOCPKT ioctl命令来设置这种模式。

Solaris和其他平台相比较，具体的打包模式有所不同。在Solaris中，读取PTY主设备的进程必须调用getmsg从流首取得消息，这是因为pckt模块将一些事件转化为无数据的STREAMS消息。在其他平台，每一次对PTY主设备的read操作都会在返回状态字节后跟随可选的数据。

无论实现细节如何，打包模式的目的是，当PTY从设备之上的行规程模块出现以下事件时，通知进程从PTY主设备读取数据：读队列被刷新；写队列被刷新；输出被停止（如：Ctrl+S）；输出重新开始；XON/XOFF流控制被关闭后重新启用；XON/XOFF流控制被启用后重新关闭。这些事件被rlogin客户进程和rlogind服务器进程等使用。 705

2. 远程模式

PTY主设备可以用TIOCREMOTE ioctl命令将PTY从设备设置成远程模式。虽然FreeBSD 5.2.1、Mac OS X 10.3和Solaris 9使用同样的命令来打开或关闭这个特性，但是在Solaris中，ioctl的第三个参数是一个整型数，而在FreeBSD和Mac OS X中是一个指向整型数的指针。(Linux 2.4.22不支持这一命令。)

当PTY主设备将PTY从设备设置成这种模式时，它通知PTY从设备之上的行规程模块对从主设备接收到的任何数据都不进行任何处理，不管从设备termios结构中的规范或非规范标志是否设置。远程模式适用于窗口管理器这种进行自己的行编辑的应用程序。

3. 窗口大小变化

PTY主设备之上的进程可以用TIOCSWINSZ ioctl命令来设置从设备的窗口大小。如果新的大小和旧的不同，SIGWINCH信号将被发送到PTY从设备的前台进程组。

4. 信号发生

读、写PTY主设备的进程可以向PTY从设备的进程组发送信号。在Solaris 9中，可以用TIOCSIGNAL ioctl命令做到这一点，该命令的第三个参数就是信号编号。在FreeBSD 5.2.1和Mac OS X 10.3中，用TIOCSIG ioctl来做到这一点，它的第三个参数是指向信号编号值的指针。(Linux 2.4.22不支持这一ioctl命令。)

19.8 小结

本章开始部分扼要叙述了如何使用伪终端并观察了某些应用实例。接着，分析说明了在本书讨论的四种平台上打开伪终端所需的代码，然后用此代码提供了通用的pty_fork函数，它可用于多种不同的应用。该函数是小程序pty的基础。我们使用pty揭示了伪终端的许多属性。

伪终端在大多数UNIX系统中每天都被用来进行网络登录。我们检查了伪终端的许多其他用途，从script程序到使用批处理脚本来驱动交互式程序等。

习题

19.1 当用telnet或rlogin远程登录到一个BSD系统上时，像我们在19.3.2节讨论过的那样，PTY从设备的所有权和权限被设置。该过程是如何发生的? 706

19.2 在BSD系统上修改程序清单19-2中的grantpt函数，使之调用一个设置用户ID程序来改变PTY从设备的所有权和权限（类似于Solaris中的grantpt函数所做的那样）。

19.3 使用pty程序来决定你的系统用来初始化PTY从设备的termios结构和winsize结构的值。

19.4 重写loop函数（见程序清单19-6），使之成为使用select或poll的单个进程。

19.5 在子进程中，pty_fork返回后，标准输入、标准输出和标准出错都以读写模式打开。能够将标准输入变成只读，另两个变成只写吗？

19.6 在图19-7中，指出哪个进程组是前台的，哪个进程组是后台的，并指出会话首进程。

19.7 在图19-7中，当键入文件终止符时，进程终止的顺序是什么？如果可能的话，用进程会计信息验证之。

19.8 script(1)程序通常在输出文件头增加一行说明它的开始时间，在输出文件末尾增加一行说明它的结束时间。将这个特性加到本章简单的shell脚本中。

19.9 解释为什么在下面的例子中，文件data的内容被输出到终端上，在什么时候程序ttyname只产生输出而从不读取输入。

```
$ cat data                      一个有两行的文件
hello,
world
$ pty -i < data ttyname         -i表示忽略stdin的文件结束标志
hello,                          这两行来自何处?
world
fd 0: /dev/ttyp5                我们期望ttyname输出这三行
fd 1: /dev/ttyp5
fd 2: /dev/ttyp5
```

19.10 编写一个调用pty_fork的程序，该程序用子进程exec另一个你必须编写的程序。后面这道新程序能够捕获SIGTERM和SIGWINCH。当捕获到信号时，要打印出有关消息，并且对于后一种信号，还要打印终端窗口大小。然后让父进程用19.7节描述过的ioctl向PTY从设备的进程组发送SIGTERM信号。从PTY从设备读回消息，验证捕获的信号。接下来由父进程设置PTY从设备窗口的大小，并读回PTY从设备的输出。让父进程退出并确定是否PTY从设备进程也终止了；如果它也终止了，请解释它何以终止。

707

第20章

数据库函数库

20.1 引言

20世纪80年代早期，UNIX系统被认为不是一个适合运行多用户数据库系统的环境（见Stonebraker[1981]和Weinberger[1982]）。早期的系统（如V7），因为没有提供任何形式的IPC机制（除了半双工管道），也没有提供任何形式的字节范围锁机制，所以确实不适合运行多用户数据库系统。但是，这些缺陷中的大多数都已得到纠正。到了20世纪80年代后期，UNIX系统已为运行可靠的、多用户的数据库系统提供了一个适合的环境。自那时以来很多商业公司都已提供这种数据库系统。

本章将开发一个简单的、多用户数据库的C函数库。调用此函数库提供的C语言函数，其他程序可以读取和存储数据库中的记录。这个C函数库通常只是一个完整的数据库系统的一部分，这里并不开发其他部分（如查询语言等），关于其他部分可以参阅专门介绍数据库系统的教科书。我们感兴趣的是数据库函数库与UNIX系统的接口，以及这些接口与前面各章节所涉及主题的关系（如14.3节的字节范围锁）。

20.2 历史

dbm(3)是在UNIX系统中很流行的数据库函数库，它由Ken Thompson开发，使用了动态散列结构。最初，它与V7一起提供，并出现在所有BSD版本中，也包含在SVR4的BSD兼容函数库中[AT&T 1990c]。BSD的开发者扩充了dbm函数库，并将它称为ndbm。ndbm函数库包括在BSD和SVR4中。ndbm函数被标准化后成为Single UNIX Specification的XSI扩展部分。

709

Seltzer和Yigit[1991]中详细介绍了dbm函数库使用的动态散列算法的历史，以及这个库的其他实现方法（例如，dbm函数库的GNU版本gdbm）。但是，所有这些实现的一个根本缺点是：它们都不支持多个进程对数据库的并发更新。它们都没有提供并发控制（如记录锁）。

4.4BSD提供了一个新的库——db(3)，该库支持三种不同的访问模式：面向记录、散列和B-树。同样，db(3)也没有提供并发控制（这一点在db(3)手册页的BUGS部分中说得很清楚）。

> Sleepycat Software (http://www.sleepycat.com) 提供了几个db函数库版本，它们支持并发访问、锁和事务。

绝大部分商用数据库函数库提供多进程同时更新数据库所需要的并发控制。这些系统一般都使用14.3节中介绍的建议记录锁，但是，它们也常常实现自己的锁原语，以避免为获得一把无竞争的锁而需的系统调用开销。这些商用系统通常用B+树[Comer 1979]，或者某种动态散列技术（例如线性散列[Litwin 1980]或者可扩展的散列[Fagin et al. 1979]）来实现数据库。

表20-1列出了本书说明的四种操作系统中常用的数据库函数库。注意在Linux上，gdbm库既支持dbm函数库，又支持ndbm函数库。

表20-1　多种平台支持的数据库函数库

函数库	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
dbm			gdbm		•
ndbm	XSI	•	gdbm	•	•
db		•	•	•	•

20.3　函数库

本章开发的函数库类似于ndbm函数库，但增加了并发控制机制，从而允许多进程同时更新同一数据库。本节将述数据库函数库的C语言接口，下一节再讨论其实际的实现。

当打开数据库时，通过返回值得到代表数据库的句柄（一个难以理解的指针（opaque pointer））。此句柄将作为参数传递给其他数据库函数。

```
#include "apue_db.h"

DBHANDLE db_open(const char *pathname, int oflag, ... /* int mode */);
                    返回值：若成功则返回数据库句柄，若出错则返回NULL

void db_close(DBHANDLE db);
```

710

如果db_open成功返回，则将建立两个文件：*pathname.idx*和*pathname.dat*，*pathname.idx*是索引文件，*pathname.dat*是数据文件。open的第二个参数*oflag*（见3.3节）指定这些文件的打开模式（只读、读/写或如果文件不存在则创建等）。如果需要建立新的数据库文件，*mode*将作为第三个参数传递给open（文件访问权限）。

当不再使用数据库时，调用db_close来关闭数据库。db_close将关闭索引文件和数据文件，并释放数据库使用过程中分配到的所有用于内部缓冲的存储空间。

当向数据库中加入一条新的记录时，必须指明此记录的键，以及与此键相联系的数据。如果此数据库存储的是人事信息，键可以是雇员ID号，数据可以是此雇员的姓名、地址、电话号码以及受聘日期等等。我们的实现要求每条记录的键必须是唯一的（例如，两条雇员记录不能有同样的雇员ID号）。

```
#include "apue_db.h"

int db_store(DBHANDLE db, const char *key, const char *data,
             int flag);
                    返回值：若成功则返回0，若出错则返回非0值（见下）
```

参数*key*和*data*是由null结束的字符串。它们可以包含除了null外的任何字符，如换行符。

*flag*参数只能是DB_INSERT（加一条新记录）、DB_REPLACE（替换一条已有的记录）或DB_STORE（加一条新记录或替换一条已有的记录，只要合适无论哪一种都可以）。这三个常数定义在apue_db.h头文件中。如果使用DB_INSERT或DB_STORE，并且记录并不存在，则加入一条新记录；如果使用DB_REPLACE或DB_STORE，并且该记录已经存在，则用新记录替

换已存在的原记录；如果使用DB_REPLACE，而记录不存在，则errno设置为ENOENT，返回值为-1，并且不加入新记录；如果使用DB_INSERT，而记录已经存在，则不加入新记录，返回值为1，在这里，返回1以区别于一般的出错返回（-1）。

通过提供键*key*可以从数据库中取出一条记录。

```
#include "apue_db.h"

char *db_fetch(DBHANDLE db, const char *key);
```

返回值：若成功则返回指向数据的指针，若记录没有找到则返回NULL

如果记录找到了，则返回指向按键*key*存放的数据的指针。通过指明键*key*，也可以在数据库中删除一条记录。

```
#include "apue_db.h"

int db_delete(DBHANDLE db, const char *key);
```

返回值：若成功则返回0，若记录没有找到则返回-1

711

除了通过键访问数据库外，也可以一条一条记录地访问数据库。为此，首先调用db_rewind回滚到数据库的第一条记录，然后在每一次循环中调用db_nextrec，顺序地读每条记录。

```
#include "apue_db.h"

void db_rewind(DBHANDLE db);

char *db_nextrec(DBHANDLE db, char *key);
```

返回值：若成功则返回指向数据的指针，若到达数据库的结尾则返回NULL

如果*key*是非空的指针，则db_nextrec将当前记录的键复制到*key*所指向的存储区中。

db_nextrec不保证记录访问的次序，只保证每一条记录被访问恰好一次。纵使顺序地存储三条键分别为A、B、C的记录，也无法确定db_nextrec将按什么顺序返回这三条记录。它可能按B、A、C的顺序返回，也可能按其他顺序，实际的顺序由数据库的实现决定。

这七个函数提供了数据库函数库的接口。接下来介绍实现。

20.4 实现概述

大多数访问数据库的函数库使用两个文件来存储信息：一个索引文件和一个数据文件。索引文件包含索引值（键）和指向数据文件中对应数据记录的指针。有许多技术可用来组织索引文件，以提高按键查询的速度和效率，散列法和B+树是两种常用的技术。我们采用固定大小的散列表来组织索引文件结构，并采用链表法解决散列冲突。在介绍db_open时，曾提到将创建两个文件：一个以.idx为后缀的索引文件和一个以.dat为后缀的数据文件。

这里将键和索引以由null结尾的字符串形式存储——它们不能包含任意的二进制数据。有些数据库系统用二进制形式存储数值数据（例如，用1、2或4个字节存储一个整数）以节省存储空间，这样一来使函数复杂化，也使数据库文件在不同的平台间移植比较困难。例如，网络上有两个系统使用不同的二进制格式存储整数，如果想要这两个系统都能够访问数据库就必须解决不同存储格式的问题（今天不同体系结构的系统共享文件已经很常见了）。按照字符串形式

存储所有的记录（包括键和数据）能使这一切变得简单，尽管这确实需要使用更多的磁盘空间，但这是为可移植性付出的很小代价。

db_store要求对于每个键，只有一条对应的记录。有些数据库系统允许多条记录使用同样的键，并提供方法访问与一个键相关的所有记录。另外，由于只有一个索引文件，这意味着每条数据记录只能有一个键（不支持第二个键）。有些数据库系统允许一条记录拥有多个键，并且对每一个键使用一个索引文件，当加入或删除一条记录时，要对所有的索引文件进行相应的更新。（一个有多个索引文件的例子是雇员库文件，可以将雇员ID号作为索引的键，也可以将雇员的社会保险号作为索引键。由于雇员的名字并不保证唯一，所以名字不能作为键。）

712

图20-1是数据库实现的基本结构。索引文件由三部分组成：空闲链表指针、散列表和索引记录。在图20-1中，所有指针（*ptr*）字段中实际存储的是以ASCII码数字形式记录的文件中的偏移量。

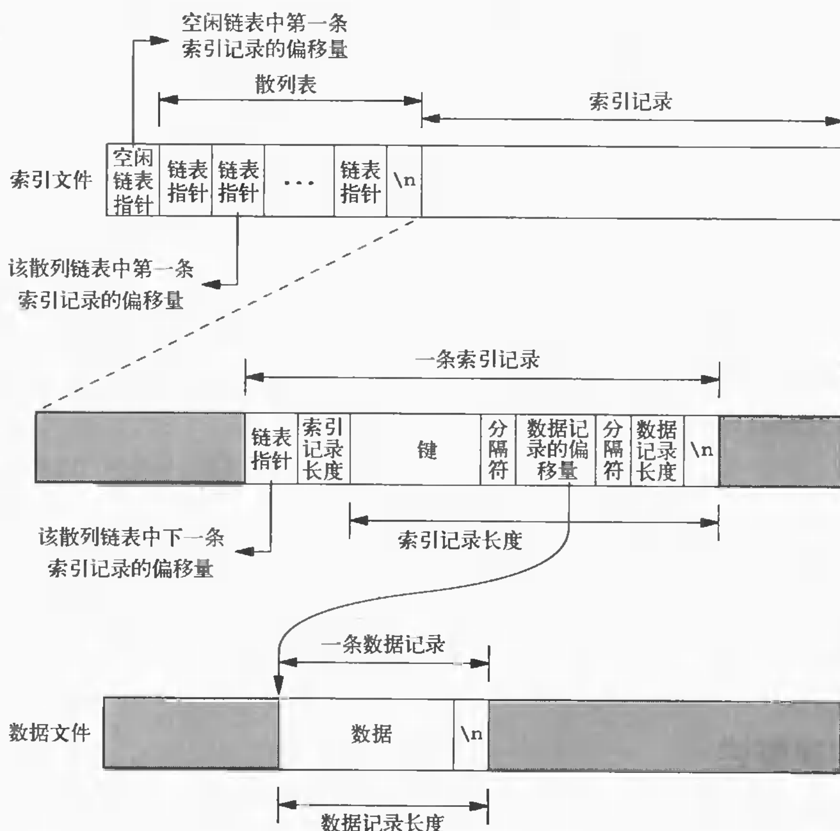


图20-1 索引文件和数据文件结构

当给定一个键要在数据库中寻找一条记录时，db_fetch根据该键计算散列值，由此散列值可确定散列表中的一条散列链（链表指针字段可以为0，表示一条空的散列链）。沿着这条散列链，可以找到所有具有该散列值的索引记录。当遇到一条索引记录的链表指针字段为0时，表示到达了此散列链的末尾。

713

下面来看一个实际的数据库文件。程序清单20-1中的程序建立了一个新的数据库，并且写入了三条记录。由于所有的字段都以ASCII字符的形式存储在数据库中，所以可以用任何标准的UNIX系统工具来查看实际的索引文件和数据文件。

```
$ ls -l db4.*
-rw-r--r--  1 sar        28 Oct 19 21:33 db4.dat
-rw-r--r--  1 sar        72 Oct 19 21:33 db4.idx
$ cat db4.idx
   0  53  35   0
   0  10Alpha:0:6
   0  10beta:6:14
  17  11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

为了使这个例子简单，将每个**指针**（*ptr*）字段的大小定为4个ASCII字符，将散列链的条数定为3。由于每一个*ptr*记录的是一个文件偏移量，所以4个ASCII字符限制了一个索引文件或数据文件的大小最多只能为10 000字节。当在20.9节做数据库系统的性能测试时，将*ptr*字段的大小设为6个字符（这样文件大小可以达到1 000 000字节），将散列链数设为100以上。

程序清单20-1　建立一个数据库并向其中写三条记录

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>

int
main(void)
{
    DBHANDLE    db;

    if ((db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) == NULL)
        err_sys("db_open error");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

索引文件的第一行为：

```
0  53  35   0
```

分别为空闲链表指针（0表示空闲链表为空），和三个散列链的指针：53、35和0。下一行：

```
0  10Alpha:0:6
```

显示了一条索引记录的结构。第一个4字符字段（0）为链表指针，表示这一条记录是此散列链的最后一条。下一个4字符字段（10）为**索引记录长度**（*idx len*），表示此索引记录剩余部分的长度。用两个read操作来读取一条索引记录：第一个read读取这两个固定长度的字段（*chain ptr*和*idx len*），然后再根据*idx len*来读取后面的不定长部分。剩下的三个字段为：**键**（*key*）、**数据记录的偏移量**（*dat off*）和**数据记录的长度**（*dat len*），这三个字段用**分隔符**（*sep*）隔开，在这里使用的是冒号。由于此三个字段都是不定长的，所以需要一个专门的分隔符，而且这个分隔符不能出现在键中。最后用一个\n（换行符）结束这一条索引记录。由于在idx len中已经

有了记录的长度，所以这个换行符并不是必需的，加上换行符是为了把各条索引记录分开，这样就可以用标准的UNIX系统工具（如cat和more）来查看索引文件。键（*key*）是将记录加入数据库时指定的值。数据记录在数据文件中的偏移量为0，长度为6。从数据文件中可看到数据记录确实从0开始，长度为6个字节。（与索引文件一样，这里自动在每条数据记录的后面加上一个换行符，以便于使用UNIX系统工具。在调用db_fetch时，此换行符不作为数据返回。）

如果在这个例子中跟踪三条散列链，可以看到第一条散列链上第一条记录的偏移量是53（gamma），这条链上下一条记录的偏移量为17（alpha），并且是这条链上的最后一条记录。第二条散列链上的第一条记录的偏移量是35（beta），且是此链上最后一条记录。第三条散列链为空。

请注意索引文件中键的顺序和数据文件中对应数据记录的顺序与程序清单20-1的程序中调用db_store的顺序相同。由于在调用db_open时使用了O_TRUNC标志，索引文件和数据文件都被截断，整个数据库相当于重新初始化。在这种情形下，db_store将新的索引记录和数据记录添加到对应的文件末尾。后面将看到db_store也可以重复使用这两个文件中因删除记录而生成的空间。

对于索引文件使用固定大小的散列表是一个折中，当每个散列链都不太长时，这种方法能保证快速地查找。我们的目的是能够快速地查找任一键，同时又不使用太复杂的数据结构，如B-树或动态可扩充散列表。动态可扩充散列表的优点是能保证仅用两次磁盘操作就能找到数据记录（详见Litwin [1980] 或 Fagin et al. [1979]）。B-树能够用（已排序的）键的顺序来遍历数据库（采用散列表的db_nextrec函数就做不到这一点）。

714～715

20.5 集中式或非集中式

当有多个进程访问同一数据库时，有两种方法可实现库函数：

(1) 集中式。由一个进程作为数据库管理者，所有的数据库访问工作由此进程完成。其他进程通过IPC机制与此中心进程进行联系。

(2) 非集中式。每个库函数独立申请并发控制（加锁），然后自己调用I/O函数。

使用这两种技术的数据库系统都有。如果有适当的加锁例程，因为避免了使用IPC，那么非集中式方法一般要快一些。图20-2描绘了集中式方法的操作。

图中特意表示出IPC像绝大多数UNIX系统的消息传送一样需要经过操作系统内核（15.9节中说明的共享存储不需要这种经过内核的复制）。可以看出，在集中式方法下，中心控制进程将记录读出，然后通过IPC机制将数据传送给请求进程，这是这种设计的不足之处。注意，集中式数据库管理进程是唯一对数据库文件进行I/O操作的进程。

集中式的优点是能够根据需要来对操作模式进行调整。例如，可以通过中心进程给不同的进程赋予不同的优先级，这会影响到中心进程对I/O操作的调度。而用非集中式方法则很难做到这一点。在这种情况下只能依赖于操作系统内核的磁盘I/O调度策略和加锁策略（也就是说，当三个进程同时等待一个锁开锁时，哪个进程下一个得到锁？）。

集中式方法的另一个优点是，复原要比非集中式方法容易。在集中式方法中，所有状态信息都集中存放在一处，所以如若杀死了数据库进程，那么只需在该处查看以识别出需要解决的未完成事务，然后将数据库恢复到一致状态。

图20-3描绘了非集中式方法，本章的实现就是采用这种方法。

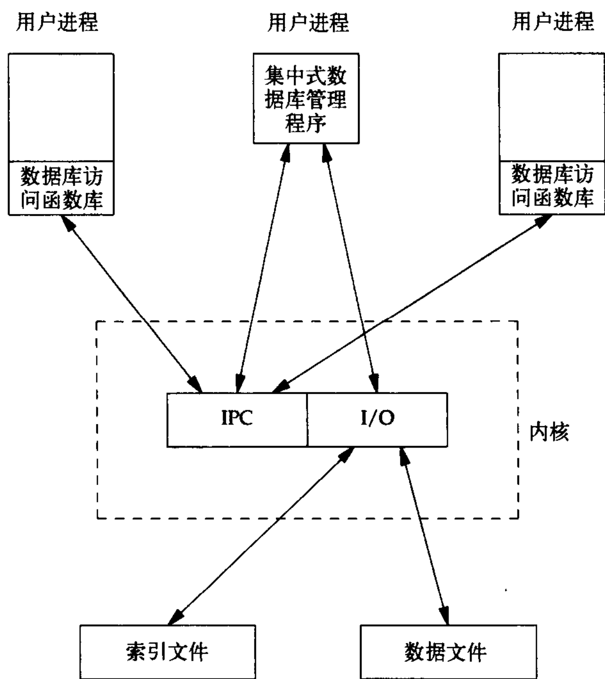


图20-2 集中式数据库访问

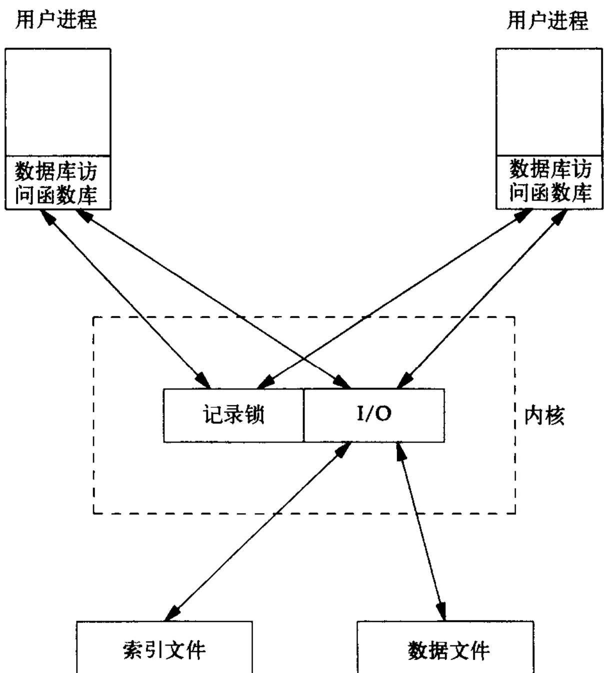


图20-3 非集中式数据库访问

调用数据库库函数执行I/O操作的用户进程是合作进程，它们使用字节范围锁机制来实现并发访问控制。

716~717

20.6 并发

由于很多系统的实现都采用两个文件（一个索引文件和一个数据文件）的方式，所以这里也特地使用这种实现方法，这要求能够控制对两个文件的加锁。有很多方法可用来对两个文件进行加锁。

1. 粗锁

最简单的加锁方法是将这两个文件中的一个作为整个数据库的锁，并要求调用者在对数据库进行操作前必须获得这个锁，这种加锁方式称为*粗锁*（coarse-grained locking）。例如，可以认为一个进程对索引文件的0字节加了读锁后，就能读整个数据库；一个进程对索引文件的0字节加了写锁后，就能写整个数据库。可以使用UNIX系统的字节范围锁机制来控制每次可以有多个读进程，而只能有一个写进程（见表14-2）。db_fetch和db_nextrec函数将要求具有读锁，而db_delete、db_store以及db_open则要求具有写锁。（db_open要求写锁的原因是：如果要创建新文件的话，要在索引文件前端建立空的空闲链表以及散列链表。）

粗锁的问题是它限制了最大程度的并发。用粗锁时，当一个进程向一条散列链中加入一条记录时，其他进程无法访问另一条散列链上的记录。

2. 细锁

下面用称为*细锁*（fine-grained locking）的方法来改进粗锁以提高并发度。首先，要求一个读进程或写进程在操作一条记录前必须先获得此记录所在散列链的读锁或写锁，允许对同一条散列链同时可以有多个读进程，而只能有一个写进程。其次，一个写进程在操作空闲链表（如db_delete或db_store）前，必须获得空闲链表的写锁。最后，当db_store向索引文件或数据文件末尾追加一条新记录时，必须获得对应文件相应区域的写锁。

期望细锁能比粗锁提供更高的并发度，20.9节将给出一些实际的比较测试结果。20.8节给出了采用细锁实现的源代码，并详细讨论了锁的实现（粗锁是实现的简化）。

在源代码中，直接调用了read，readv，write和writev，而没有使用标准I/O函数库。虽然使用标准I/O函数库也可以使用字节范围锁，但是需要非常复杂的缓冲管理。例如，当另一个进程在5分钟之前修改了数据，不希望fgets返回10分钟前读入标准I/O缓冲的数据。

718

这里对并发进行讨论，所依据的是对数据库函数库的简单需求，商业系统一般有更多的需要。关于并发更多的细节可以参见Date[2004]的第16章。

20.7 构造函数库

数据库的函数库由两个文件构成，它们是：公用的C头文件以及一个C源文件。可以用下列命令构造一静态函数库。

```
gcc -I../include -Wall -c db.c
ar rsv libapue_db.a db.o
```

因为在数据库函数库中使用了一些我们自己的公共函数，所以希望与libapue_db.a相链接的应用程序也需要与libapue.a相链接。

另一方面，如果想构建数据库函数库的动态共享库版本，那么可使用下列命令：

```
gcc -I../include -Wall -fPIC -c db.c
gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 \
    -L../lib -lapue -lc db.o
```

构建成的共享库libapue_db.so.1需放置在动态链接/装入程序（dynamic linker/loader）能够找到的公用目录中。另一个替代的方法是：将共享库放置在私有目录中，修改LD_LIBRARY_PATH环境变量，使动态链接/装入程序的搜索路径包含该私有目录。

在不同平台上构建共享库的步骤会有所不同。这里所给出的步骤是在带GNU C编译器的Linux系统中进行的。

20.8 源代码

本节对所编写的数据库函数库源代码进行解说，先从头文件apue_db.h开始。函数库源代码以及调用此函数库的所有应用程序都包含这一头文件。

从此处开始，实例程序的编排方式在很多方面与前面的有所不同。首先，因为源代码较长，为此加了行号，这使得联系相应的源代码进行讨论更加方便。其次，对源代码的说明紧随其后。

这种风格受到John Lions对UNIX V6源代码注释一书[Lions 1977，1996]的影响，使得解释说明大量源代码更为简易。

注意，这里对空白行不编号。虽然某些工具（例如pr(1)）的正常操作与这些空白行是有关的，但是此处对它们并无任何兴趣。

719

```
 1  #ifndef _APUE_DB_H
 2  #define _APUE_DB_H

 3  typedef   void *  DBHANDLE;

 4  DBHANDLE  db_open(const char *, int, ...);
 5  void      db_close(DBHANDLE);
 6  char     *db_fetch(DBHANDLE, const char *);
 7  int       db_store(DBHANDLE, const char *, const char *, int);
 8  int       db_delete(DBHANDLE, const char *);
 9  void      db_rewind(DBHANDLE);
10  char     *db_nextrec(DBHANDLE, char *);

11  /*
12   * Flags for db_store().
13   */
14  #define DB_INSERT      1    /* insert new record only */
15  #define DB_REPLACE     2    /* replace existing record */
16  #define DB_STORE       3    /* replace or insert */

17  /*
18   * Implementation limits.
19   */
20  #define IDXLEN_MIN    6    /* key, sep, start, sep, length, \n */
21  #define IDXLEN_MAX 1024    /* arbitrary */
22  #define DATLEN_MIN    2    /* data byte, newline */
23  #define DATLEN_MAX 1024    /* arbitrary */

24  #endif /* _APUE_DB_H */
```

[1-3]　使用符号_APUE_DB_H以保证只包含该头文件一次。DBHANDLE类型表示对数据库的一个有效引用，用于隔离应用程序和数据库的实现细节。数据库函数库

向应用程序提供DBHANDLE类型，而标准I/O库则向应用程序提供FILE结构，两者相比非常相似。

[4-10]　接着，声明了数据库函数库公用函数的原型。因为使用函数库的应用程序包含了此文件，所以这里不再声明函数库私有函数的原型。

[11-24]　定义了可以传送给db_store函数的合法标志，其后是实现的基本限制。为支持更大的数据库可以更改这些限制。

最小索引记录长度由IDXLEN_MIN指定。这表示1字节键、1字节分隔符、1字节起始偏移量、另一个1字节分隔符、1字节长度和终止换行符。（回忆图20-1中索引记录的格式。）一条索引记录通常长于IDXLEN_MIN字节，这只是最小长度。

720

下一个文件是db.c，它是库函数的C源文件。为简化起见，将所有函数都放在一个文件中。这样处理的优点是只要将私有函数声明为static，就可对外将它隐蔽起来。

```
 1   #include "apue.h"
 2   #include "apue_db.h"
 3   #include <fcntl.h>     /* open & db_open flags */
 4   #include <stdarg.h>
 5   #include <errno.h>
 6   #include <sys/uio.h>   /* struct iovec */

 7   /*
 8    * Internal index file constants.
 9    * These are used to construct records in the
10    * index file and data file.
11    */
12   #define IDXLEN_SZ      4     /* index record length (ASCII chars) */
13   #define SEP           ':'    /* separator char in index record */
14   #define SPACE         ' '    /* space character */
15   #define NEWLINE       '\n'   /* newline character */

16   /*
17    * The following definitions are for hash chains and free
18    * list chain in the index file.
19    */
20   #define PTR_SZ         6     /* size of ptr field in hash chain */
21   #define PTR_MAX   999999     /* max file offset = 10**PTR_SZ - 1 */
22   #define NHASH_DEF    137     /* default hash table size */
23   #define FREE_OFF       0     /* free list offset in index file */
24   #define HASH_OFF PTR_SZ      /* hash table offset in index file */

25   typedef unsigned long  DBHASH; /* hash values */
26   typedef unsigned long  COUNT;  /* unsigned counter */
```

[1-6]　由于使用了一些私有函数库中的函数，所以程序中包含了apue.h。当然，apue.h也包含若干标准头文件，包括<stdio.h>和<unistd.h>。因为db_open函数使用由<stdarg.h>定义的可变参数函数，所以程序中也包含了<stdarg.h>。

[7-26]　索引记录的长度指定为IDXLEN_SZ。用某些字符（例如冒号、换行符）作为数据库中的分隔符。当删除一条记录时，在其中全部填入空格符。

其中一些定义为常量的值也可定义为变量，只是这样会使实现复杂一些。例如，设定散列表的大小为137记录项，也许更好的方法是让db_open的调用者根据预期的数据库大小通过参数来设定这个值，然后将该值存储在索引文件的最前面。

721

```
27  /*
28   * Library's private representation of the database.
29   */
30  typedef struct {
31    int     idxfd;  /* fd for index file */
32    int     datfd;  /* fd for data file */
33    char   *idxbuf; /* malloc'ed buffer for index record */
34    char   *datbuf; /* malloc'ed buffer for data record*/
35    char   *name;   /* name db was opened under */
36    off_t   idxoff; /* offset in index file of index record */
37                    /* key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
38    size_t idxlen;  /* length of index record */
39                    /* excludes IDXLEN_SZ bytes at front of record */
40                    /* includes newline at end of index record */
41    off_t   datoff; /* offset in data file of data record */
42    size_t datlen;  /* length of data record */
43                    /* includes newline at end */
44    off_t   ptrval; /* contents of chain ptr in index record */
45    off_t   ptroff; /* chain ptr offset pointing to this idx record */
46    off_t   chainoff; /* offset of hash chain for this index record */
47    off_t   hashoff;  /* offset in index file of hash table */
48    DBHASH nhash;     /* current hash table size */
49    COUNT   cnt_delok;    /* delete OK */
50    COUNT   cnt_delerr;   /* delete error */
51    COUNT   cnt_fetchok;  /* fetch OK */
52    COUNT   cnt_fetcherr; /* fetch error */
53    COUNT   cnt_nextrec;  /* nextrec */
54    COUNT   cnt_stor1;    /* store: DB_INSERT, no empty, appended */
55    COUNT   cnt_stor2;    /* store: DB_INSERT, found empty, reused */
56    COUNT   cnt_stor3;    /* store: DB_REPLACE, diff len, appended */
57    COUNT   cnt_stor4;    /* store: DB_REPLACE, same len, overwrote */
58    COUNT   cnt_storerr;  /* store error */
59  } DB;
```

[27-48] 在DB结构中记录一个打开数据库的所有信息。db_open函数返回DB结构的指针DBHANDLE值，这个指针被用于其他所有函数，而该结构本身则不面向调用者。因为在数据库中是以ASCII码形式存放指针和长度，所以要将这些ASCII码变换为数字值，然后存放在DB结构中。DB结构中也存放散列表长度，虽然一般而言，这是定长的，但也有可能为增强该函数库，允许调用者在创建数据库时指定该长度（见习题20.7）。

[49-59] DB结构的最后10个字段对成功和不成功的操作计数。如果想要分析数据库的性能，则可编写一个函数返回这些统计值。但目前仅保持这些计数器，并未编写此种函数。

722

```
60  /*
61   * Internal functions.
62   */
63  static DB     *_db_alloc(int);
64  static void    _db_dodelete(DB *);
65  static int     _db_find_and_lock(DB *, const char *, int);
66  static int     _db_findfree(DB *, int, int);
67  static void    _db_free(DB *);
68  static DBHASH  _db_hash(DB *, const char *);
69  static char   *_db_readdat(DB *);
70  static off_t   _db_readidx(DB *, off_t);
```

```
71   static off_t   _db_readptr(DB *, off_t);
72   static void    _db_writedat(DB *, const char *, off_t, int);
73   static void    _db_writeidx(DB *, const char *, off_t, int, off_t);
74   static void    _db_writeptr(DB *, off_t, off_t);

75   /*
76    * Open or create a database.  Same arguments as open(2).
77    */
78   DBHANDLE
79   db_open(const char *pathname, int oflag, ...)
80   {
81       DB          *db;
82       int         len, mode;
83       size_t      i;
84       char        asciiptr[PTR_SZ + 1],
85                   hash[(NHASH_DEF + 1) * PTR_SZ + 2];
86                       /* +2 for newline and null */
87       struct stat statbuff;

88       /*
89        * Allocate a DB structure, and the buffers it needs.
90        */
91       len = strlen(pathname);
92       if ((db = _db_alloc(len)) == NULL)
93           err_dump("db_open: _db_alloc error for DB");
```

[60-74] 选择用db_开头来命名所有用户可调用（公用）的库函数，用_db_开头来命名内部（私有）函数。公用函数在函数库头文件apue_db.h中声明。内部函数声明为static，所以只有同一文件中的其他函数才能调用它们（该文件包含函数库实现）。

[75-93] db_open函数的参数与open(2)相同。如果调用者想要创建数据库文件，那么用可选择的第三个参数指定文件权限。db_open函数打开索引文件和数据文件，在必要时初始化索引文件。该函数调用_db_alloc来为DB结构分配空间，并初始化此结构。

723

```
94       db->nhash   = NHASH_DEF;/* hash table size */
95       db->hashoff = HASH_OFF; /* offset in index file of hash table */
96       strcpy(db->name, pathname);
97       strcat(db->name, ".idx");

98       if (oflag & O_CREAT) {
99           va_list ap;

100          va_start(ap, oflag);
101          mode = va_arg(ap, int);
102          va_end(ap);

103          /*
104           * Open index file and data file.
105           */
106          db->idxfd = open(db->name, oflag, mode);
107          strcpy(db->name + len, ".dat");
108          db->datfd = open(db->name, oflag, mode);
109      } else {
110          /*
111           * Open index file and data file.
112           */
113          db->idxfd = open(db->name, oflag);
```

```
114         strcpy(db->name + len, ".dat");
115         db->datfd = open(db->name, oflag);
116     }

117     if (db->idxfd < 0 || db->datfd < 0) {
118         _db_free(db);
119         return(NULL);
120     }
```

[94-97]　继续初始化DB结构。调用者传入的路径名指定数据库文件名的前缀。添加后缀.idx以构成数据库索引文件的名字。

[98-108]　如果调用者想要创建数据库文件，那么使用<stdarg.h>中的可变参数函数以找到可选的第3个参数，然后，使用open来创建和打开索引文件和数据库文件。注意，数据文件的文件名与索引文件以同样的前缀开始，只是后缀为.dat。

[109-116]　如果调用者没有指定O_CREAT标志，那么正在打开现有的数据库文件。此时，只用两个参数调用open。

[117-120]　如果在打开或创建任一数据库文件时出错，则调用_db_free清除DB结构，然后对调用者返回NULL。如果一个文件open成功，另一个失败，_db_free将关闭该打开的文件描述符。很快就会见到这一操作。

724

```
121     if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
122         /*
123          * If the database was created, we have to initialize
124          * it.  Write lock the entire file so that we can stat
125          * it, check its size, and initialize it, atomically.
126          */
127         if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
128             err_dump("db_open: writew_lock error");

129         if (fstat(db->idxfd, &statbuff) < 0)
130             err_sys("db_open: fstat error");

131         if (statbuff.st_size == 0) {
132             /*
133              * We have to build a list of (NHASH_DEF + 1) chain
134              * ptrs with a value of 0.  The +1 is for the free
135              * list pointer that precedes the hash table.
136              */
137             sprintf(asciiptr, "%*d", PTR_SZ, 0);
```

[121-130]　如果正在建立数据库，则必须正确地加锁。考虑两个进程试图同时建立同一个数据库的情况。假设第一个进程运行到调用fstat，并且在fstat返回后被内核阻塞。这时第二个进程调用db_open，发现索引文件的长度为0，于是初始化空闲链表和散列链表，接着第二个进程向数据库中添加了一条记录。此时第二个进程被阻塞，第一个进程继续运行，它发现索引文件的长度为0（因为第一个进程调用fstat在前，然后第二个进程再初始化索引文件），所以第一个进程重新初始化空闲链表和散列链表，第二个进程写入的记录就被抹去了。避免发生这种情况的方法是进行加锁，为此可以使用14.3节中的readw_lock，writew_lock和un_lock这三个宏。

[131-137]　如果索引文件的长度是0，那么该文件是刚刚被创建的，所以需要初始化它所包

含的空闲链表和散列链表指针。注意，用格式字符串%*d将数据库指针从整型变换为ASCII字符串。(在_db_writeidx和_db_writeptr中还将使用这种格式字符串。）这一格式告诉sprintf取PTR_SZ参数，用它作为下一个参数的最小字段宽度，在此处为0（因为正在创建一个新的数据库，所以这里使指针取初值为0)。其作用是强迫创建的字符串至少包含PTR_SZ个字符（在左边用空格填充)。在_db_writeidx和_db_writeptr中，将传送一个非0指针值，但是首先将验证指针值不大于PTR_MAX，以保证写入数据库的指针字符串恰好为PTR_SZ(6)个字符。

725

```
138                 hash[0] = 0;
139                 for (i = 0; i < NHASH_DEF + 1; i++)
140                     strcat(hash, asciiptr);
141                 strcat(hash, "\n");
142                 i = strlen(hash);
143                 if (write(db->idxfd, hash, i) != i)
144                     err_dump("db_open: index file init write error");
145             }
146             if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
147                 err_dump("db_open: un_lock error");
148         }
149         db_rewind(db);
150         return(db);
151     }

152     /*
153      * Allocate & initialize a DB structure and its buffers.
154      */
155     static DB *
156     _db_alloc(int namelen)
157     {
158         DB      *db;

159         /*
160          * Use calloc, to initialize the structure to zero.
161          */
162         if ((db = calloc(1, sizeof(DB))) == NULL)
163             err_dump("_db_alloc: calloc error for DB");
164         db->idxfd = db->datfd = -1;              /* descriptors */

165         /*
166          * Allocate room for the name.
167          * +5 for ".idx" or ".dat" plus null at end.
168          */
169         if ((db->name = malloc(namelen + 5)) == NULL)
170             err_dump("_db_alloc: malloc error for name");
```

[138-151] 继续初始化新创建的数据库。构造散列表，将它写到索引文件中。然后，解锁索引文件，清除数据库文件指针，返回DB结构指针作为句柄，以便调用者以后用于其他数据库函数。

[152-164] db_open调用函数_db_alloc为DB结构分配空间，包括一个索引缓冲和一个数据缓冲。用calloc分配存储区，用以保存DB结构，并将该区各单元全部置初值为0。这产生了一个副作用，就是将数据库文件描述符也设置为0，因此需将它们重新设置为-1，以表示它们至此还不是有效的。

[165-170] 分配空间以存放数据库文件的名字。如db_open中所说明的那样，使用缓冲区

创建文件名，通过更改名字的后缀表示是索引文件还是数据文件。

726

```
171     /*
172      * Allocate an index buffer and a data buffer.
173      * +2 for newline and null at end.
174      */
175     if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
176         err_dump("_db_alloc: malloc error for index buffer");
177     if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
178         err_dump("_db_alloc: malloc error for data buffer");
179     return(db);
180 }

181 /*
182  * Relinquish access to the database.
183  */
184 void
185 db_close(DBHANDLE h)
186 {
187     _db_free((DB *)h);  /* closes fds, free buffers & struct */
188 }

189 /*
190  * Free up a DB structure, and all the malloc'ed buffers it
191  * may point to.  Also close the file descriptors if still open.
192  */
193 static void
194 _db_free(DB *db)
195 {
196     if (db->idxfd >= 0)
197         close(db->idxfd);
198     if (db->datfd >= 0)
199         close(db->datfd);
```

[171-180] 为索引文件和数据文件的缓冲分配空间。索引缓冲和数据缓冲的大小在apue_db.h中定义。数据库函数库可以通过让这些缓冲按需要扩张来得到增强，其方法可以是记录这两个缓冲的大小，然后在需要更大的缓冲时调用realloc。最后，返回已分配到的DB结构的指针。

[181-188] db_close函数只是一个包装，它将数据库句柄转换为DB结构的指针，将其传送给_db_free函数，由该函数释放资源以及DB结构。

[189-199] db_open在打开索引文件和数据文件时如果发生错误，则调用_db_free释放资源；应用程序在结束对数据库的使用后，db_close也调用_db_free。如果数据库索引文件的文件描述符有效，那么关闭该文件；对数据文件的文件描述符也作同样处理。（回忆当在_db_alloc中分配一新的DB结构时，对每个文件描述符都赋初值−1。如果不能打开两个数据库文件中的一个，由于相应的文件描述符仍为−1，于是也就无需关闭它。）

727

```
200     if (db->idxbuf != NULL)
201         free(db->idxbuf);
202     if (db->datbuf != NULL)
203         free(db->datbuf);
204     if (db->name != NULL)
205         free(db->name);
206     free(db);
207 }
```

```
208   /*
209    * Fetch a record.  Return a pointer to the null-terminated data.
210    */
211   char *
212   db_fetch(DBHANDLE h, const char *key)
213   {
214     DB      *db = h;
215     char    *ptr;

216     if (_db_find_and_lock(db, key, 0) < 0) {
217         ptr = NULL;             /* error, record not found */
218         db->cnt_fetcherr++;
219     } else {
220         ptr = _db_readdat(db);  /* return pointer to data */
221         db->cnt_fetchok++;
222     }

223     /*
224      * Unlock the hash chain that _db_find_and_lock locked.
225      */
226     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
227         err_dump("db_fetch: un_lock error");
228     return(ptr);
229   }
```

[200-207] 接着，释放动态分配的缓冲。可以安全地将一个空指针传送给free函数，因此也就无需事先检查每个缓冲指针的值，但无论如何还是要这样做，因为只释放已分配的对象被认为是一种较好的编程风格。(并非所有释放函数都像free那样容忍差错。) 最后，释放DB结构占用的存储区。

[208-218] 函数db_fetch根据给定的键来读取一条记录。它首先调用_db_find_and_lock在数据库中查找该记录。若不能找到该记录，则将返回值（ptr）设置为NULL，并将不成功的记录搜索计数值加1。因为从_db_find_and_lock返回时，数据库索引文件是加锁的，所以先要解锁，然后再返回。

[219-229] 如果找到了记录，调用_db_readdat读相应的数据记录，并将成功的记录搜索计数值加1。在返回前，调用un_lock对索引文件解锁，然后，返回所找到记录的指针（如果没有找到所需记录，则返回NULL)。

728

```
230   /*
231    * Find the specified record.  Called by db_delete, db_fetch,
232    * and db_store.  Returns with the hash chain locked.
233    */
234   static int
235   _db_find_and_lock(DB *db, const char *key, int writelock)
236   {
237     off_t   offset, nextoffset;

238     /*
239      * Calculate the hash value for this key, then calculate the
240      * byte offset of corresponding chain ptr in hash table.
241      * This is where our search starts.  First we calculate the
242      * offset in the hash table for this key.
243      */
244     db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
245     db->ptroff = db->chainoff;

246     /*
```

```
247         * We lock the hash chain here.  The caller must unlock it
248         * when done.  Note we lock and unlock only the first byte.
249         */
250        if (writelock) {
251            if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
252                err_dump("_db_find_and_lock: writew_lock error");
253        } else {
254            if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
255                err_dump("_db_find_and_lock: readw_lock error");
256        }
257        /*
258         * Get the offset in the index file of first record
259         * on the hash chain (can be 0).
260         */
261        offset = _db_readptr(db, db->ptroff);
```

[230-237] _db_find_and_lock函数在函数库内部用于按给定的键查找记录。在搜索记录时，如果想在索引文件上加一把写锁，则将writelock参数设置为非0值；如果将writelock参数设置为0，则在搜索记录时，在索引文件上加读锁。

[238-256] 在_db_find_and_lock中准备遍历散列链。将键变换为散列值，用其计算在文件中相应散列链的起始地址（chainoff）。在遍历散列链前，等待获得锁。注意，只锁该散列链开始处的第1个字节，这种方式允许多个进程同时搜索不同的散列链，因此增加了并发性。

[257-261] 调用_db_readptr读散列链中的第一个指针。如果该函数返回0，则该散列链为空。

729

```
262        while (offset != 0) {
263            nextoffset = _db_readidx(db, offset);
264            if (strcmp(db->idxbuf, key) == 0)
265                break;       /* found a match */
266            db->ptroff = offset; /* offset of this (unequal) record */
267            offset = nextoffset; /* next one to compare */
268        }
269        /*
270         * offset == 0 on error (record not found).
271         */
272        return(offset == 0 ? -1 : 0);
273    }
274    /*
275     * Calculate the hash value for a key.
276     */
277    static DBHASH
278    _db_hash(DB *db, const char *key)
279    {
280        DBHASH      hval = 0;
281        char        c;
282        int         i;
283        for (i = 1; (c = *key++) != 0; i++)
284            hval += c * i;      /* ascii char times its 1-based index */
285        return(hval % db->nhash);
286    }
```

[262-268] while循环遍历散列链中的每一条索引记录，并比较键。调用函数_db_

readidx读取每条索引记录。它将当前记录的键填入DB结构中的idxbuf字段。如果_db_readidx返回0，则已到达散列链的最后一个记录项。

[269-273] 如果在循环后，offset为0，那么已到达散列链末端并且没有找到匹配键，于是返回−1；否则，找到了匹配记录（用break语句退出了循环），那么就返回0表示成功。此时，ptroff字段包含前一索引记录的地址，datoff包含数据记录的地址，datlen是数据记录的长度。当沿着散列链进行遍历时，必须始终跟踪当前索引记录的前一条索引记录，其中有一个指针指向当前索引记录。这一点在删除一条记录时很有用，因为必须修改当前索引记录的前一条记录的链表指针以删除当前记录。

[274-286] _db_hash根据给定的键计算散列值。它将键中的每一个ASCII字符乘以这个字符在字符串中以1开始的索引号，将这些结果加起来，除以散列表记录项数，将余数作为这个键的散列值。回忆散列表记录项数是137，它是一个素数，按Knuth [1998]，素数散列通常提供良好的分布特性。

730

```
287   /*
288    * Read a chain ptr field from anywhere in the index file:
289    * the free list pointer, a hash table chain ptr, or an
290    * index record chain ptr.
291    */
292   static off_t
293   _db_readptr(DB *db, off_t offset)
294   {
295       char    asciiptr[PTR_SZ + 1];

296       if (lseek(db->idxfd, offset, SEEK_SET) == -1)
297           err_dump("_db_readptr: lseek error to ptr field");
298       if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
299           err_dump("_db_readptr: read error of ptr field");
300       asciiptr[PTR_SZ] = 0;       /* null terminate */
301       return(atol(asciiptr));
302   }

303   /*
304    * Read the next index record.  We start at the specified offset
305    * in the index file.  We read the index record into db->idxbuf
306    * and replace the separators with null bytes.  If all is OK we
307    * set db->datoff and db->datlen to the offset and length of the
308    * corresponding data record in the data file.
309    */
310   static off_t
311   _db_readidx(DB *db, off_t offset)
312   {
313       ssize_t             i;
314       char            *ptr1, *ptr2;
315       char            asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
316       struct iovec    iov[2];
```

[287-302] _db_readptr函数读取以下三种不同链表指针中的任意一种：(1) 索引文件最开始处指向空闲链表中第一条索引记录的指针；(2) 散列表中指向散列链的第一条索引记录的指针；(3) 存放在每条索引记录开始处、指向下一条记录的指针（这里的索引记录既可以处于一条散列链中，也可以处于空闲链表中）。返回前，将指针从ASCII形式变换为长整型。此函数不进行任何加锁操作，所以其调用

者应事先做好必要的加锁。

[303-316] _db_readidx函数用于从索引文件的指定偏移量处读取索引记录。如果成功，该函数将返回链表中下一条记录的偏移量。并且该函数填充DB结构的许多字段：idxoff包含索引文件中当前记录的偏移量，ptrval包含在散列链表中下一条索引项的偏移量，idxlen包含当前索引记录的长度，idxbuf包含实际索引记录，datoff包含数据文件中该记录的偏移量，datlen包含该数据记录的长度。 731

```
317     /*
318      * Position index file and record the offset.  db_nextrec
319      * calls us with offset==0, meaning read from current offset.
320      * We still need to call lseek to record the current offset.
321      */
322     if ((db->idxoff = lseek(db->idxfd, offset,
323       offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
324         err_dump("_db_readidx: lseek error");

325     /*
326      * Read the ascii chain ptr and the ascii length at
327      * the front of the index record.  This tells us the
328      * remaining size of the index record.
329      */
330     iov[0].iov_base = asciiptr;
331     iov[0].iov_len  = PTR_SZ;
332     iov[1].iov_base = asciilen;
333     iov[1].iov_len  = IDXLEN_SZ;
334     if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
335         if (i == 0 && offset == 0)
336             return(-1);     /* EOF for db_nextrec */
337         err_dump("_db_readidx: readv error of index record");
338     }

339     /*
340      * This is our return value; always >= 0.
341      */
342     asciiptr[PTR_SZ] = 0;        /* null terminate */
343     db->ptrval = atol(asciiptr); /* offset of next key in chain */

344     asciilen[IDXLEN_SZ] = 0;     /* null terminate */
345     if ((db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
346       db->idxlen > IDXLEN_MAX)
347         err_dump("_db_readidx: invalid length");
```

[317-324] 按调用者提供的参数，查找索引文件偏移量。在DB结构中，记录该偏移量，为此即使调用者想要在当前文件偏移量处读记录（设置offset为0），仍需要调用lseek以确定当前偏移量。因为在索引文件中，索引记录决不会存放在偏移量为0处，所以可以放心地使用0表示"从当前偏移量处读"。

[325-338] 调用readv读在索引记录开始处的两个定长字段：指向下一条索引记录的链表指针和该索引记录余下部分的长度（余下部分是不定长的）。

[339-347] 变换下一记录的偏移量为整型，并存放到ptrval字段（这将作为此函数的返回值）。然后将索引记录的长度变换为整型，并存放在idxlen字段。 732

```
348     /*
349      * Now read the actual index record.  We read it into the key
350      * buffer that we malloced when we opened the database.
```

```
351      */
352     if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
353         err_dump("_db_readidx: read error of index record");
354     if (db->idxbuf[db->idxlen-1] != NEWLINE)    /* sanity check */
355         err_dump("_db_readidx: missing newline");
356     db->idxbuf[db->idxlen-1] = 0;    /* replace newline with null */

357     /*
358      * Find the separators in the index record.
359      */
360     if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
361         err_dump("_db_readidx: missing first separator");
362     *ptr1++ = 0;                /* replace SEP with null */

363     if ((ptr2 = strchr(ptr1, SEP)) == NULL)
364         err_dump("_db_readidx: missing second separator");
365     *ptr2++ = 0;                /* replace SEP with null */

366     if (strchr(ptr2, SEP) != NULL)
367         err_dump("_db_readidx: too many separators");

368     /*
369      * Get the starting offset and length of the data record.
370      */
371     if ((db->datoff = atol(ptr1)) < 0)
372         err_dump("_db_readidx: starting offset < 0");
373     if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
374         err_dump("_db_readidx: invalid length");
375     return(db->ptrval);     /* return offset of next key in chain */
376 }
```

[348-356] 将索引记录的不定长部分读入DB结构中的idxbuf字段。该记录应以换行符结束，将该字符代换为NULL字节。如果索引文件已遭破坏，那么调用err_dump函数构造core文件后终止。

[357-367] 将索引记录划分成三个字段：键、对应数据记录的偏移量和数据记录的长度。strchr函数在给定字符串中找到首次出现的指定字符。这里要寻找的是记录中分隔字段的字符（SEP，将其定义为冒号）。

[368-376] 将数据记录的偏移量和长度变换为整型，并把它们存放在DB结构中。然后，返回散列链中下一条记录的偏移量。注意并不读数据记录，这由调用者自己完成。例如，在db_fetch中，在_db_find_and_lock按键找到索引记录前是不读取数据记录的。

733

```
377 /*
378  * Read the current data record into the data buffer.
379  * Return a pointer to the null-terminated data buffer.
380  */
381 static char *
382 _db_readdat(DB *db)
383 {
384     if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
385         err_dump("_db_readdat: lseek error");
386     if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
387         err_dump("_db_readdat: read error");
388     if (db->datbuf[db->datlen-1] != NEWLINE)    /* sanity check */
389         err_dump("_db_readdat: missing newline");
390     db->datbuf[db->datlen-1] = 0; /* replace newline with null */
391     return(db->datbuf);     /* return pointer to data record */
```

```
392  }
393  /*
394   * Delete the specified record.
395   */
396  int
397  db_delete(DBHANDLE h, const char *key)
398  {
399    DB      *db = h;
400    int     rc = 0;          /* assume record will be found */
401    if (_db_find_and_lock(db, key, 1) == 0) {
402        _db_dodelete(db);
403        db->cnt_delok++;
404    } else {
405        rc = -1;             /* not found */
406        db->cnt_delerr++;
407    }
408    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
409        err_dump("db_delete: un_lock error");
410    return(rc);
411  }
```

[377-392] 在datoff和datlen已获正确值后，_db_readdat函数将数据记录的内容读入DB结构中的datbuf字段指向的缓冲区。

[393-411] db_delete函数用于删除与给定键匹配的一条记录。调用_db_find_and_lock判断在数据库中该记录是否存在，如果存在，则调用_db_dodelete函数执行删除该记录的操作。_db_find_and_lock的第3个参数控制对散列链是加读锁，还是写锁。此处，因为可能执行更改该链表的操作，所以要加一把写锁。_db_find_and_lock返回时，这把锁仍旧存在，为此不管是否找到了所需的记录，都需要除去这把锁。

734

```
412  /*
413   * Delete the current record specified by the DB structure.
414   * This function is called by db_delete and db_store, after
415   * the record has been located by _db_find_and_lock.
416   */
417  static void
418  _db_dodelete(DB *db)
419  {
420    int     i;
421    char    *ptr;
422    off_t   freeptr, saveptr;
423    /*
424     * Set data buffer and key to all blanks.
425     */
426    for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
427        *ptr++ = SPACE;
428    *ptr = 0;    /* null terminate for _db_writedat */
429    ptr = db->idxbuf;
430    while (*ptr)
431        *ptr++ = SPACE;
432    /*
433     * We have to lock the free list.
434     */
```

```
435      if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
436          err_dump("_db_dodelete: writew_lock error");
437      /*
438       * Write the data record with all blanks.
439       */
440      _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

[412-431] _db_dodelete函数执行从数据库中删除一条记录的所有操作。(该函数也可以由db_store调用)。此函数的大部分工作仅仅是更新两个链表：空闲链表以及与键对应的散列链。当一条记录被删除后，将其键和数据记录设为空。本节后面将提到的函数db_nextrec要用到这一点。

[432-440] 调用writew_lock对空闲链表加写锁，这样能防止两个进程同时删除不同散列链上的记录时产生相互影响，因为要将被删除的记录移到空闲链表上，这将改变空闲链表指针，而一次只能有一个进程能这样做。

_db_dodelete调用函数_db_writedat清空数据记录，注意此时_db_writedat无需对数据文件加写锁，因为db_delete对这条记录的散列链已经加了写锁，这便保证不再会有其他进程能够读写该记录。

735

```
441      /*
442       * Read the free list pointer.  Its value becomes the
443       * chain ptr field of the deleted index record.  This means
444       * the deleted record becomes the head of the free list.
445       */
446      freeptr = _db_readptr(db, FREE_OFF);
447      /*
448       * Save the contents of index record chain ptr,
449       * before it's rewritten by _db_writeidx.
450       */
451      saveptr = db->ptrval;
452      /*
453       * Rewrite the index record.  This also rewrites the length
454       * of the index record, the data offset, and the data length,
455       * none of which has changed, but that's OK.
456       */
457      _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);
458      /*
459       * Write the new free list pointer.
460       */
461      _db_writeptr(db, FREE_OFF, db->idxoff);
462      /*
463       * Rewrite the chain ptr that pointed to this record being
464       * deleted.  Recall that _db_find_and_lock sets db->ptroff to
465       * point to this chain ptr.  We set this chain ptr to the
466       * contents of the deleted record's chain ptr, saveptr.
467       */
468      _db_writeptr(db, db->ptroff, saveptr);
469      if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
470          err_dump("_db_dodelete: un_lock error");
471  }
```

[441-461] 读空闲链表指针，接着修改索引记录，让这条记录中的下一条记录指针指向空闲链表的第一条记录（如果空闲链表为空，则这个链表指针置为0）。既然已经清除

了键，就用正被删除的索引记录的偏移量更新空闲链表指针，也就是使其指向当前删除的这条记录，这样就将这条删除的记录加到了空闲链表之首。空闲链表实际上很像一个后进先出的堆栈（虽然是以首次适应算法分配空闲链表项）。

没有为每个文件分别设置空闲链表。当把一个删除的索引记录加入空闲链表时，该索引记录仍指向已删除的数据记录。有更好的处理方法，但复杂性增加了。

[462-471] 修改散列链中前一条记录的指针，使其指向正删除记录之后的一条记录，这样便从散列链中撤除了要删去的记录。最后对空闲链表解锁。

736

```
472    /*
473     * Write a data record.  Called by _db_dodelete (to write
474     * the record with blanks) and db_store.
475     */
476    static void
477    _db_writedat(DB *db, const char *data, off_t offset, int whence)
478    {
479      struct iovec    iov[2];
480      static char     newline = NEWLINE;

481      /*
482       * If we're appending, we have to lock before doing the lseek
483       * and write to make the two an atomic operation.  If we're
484       * overwriting an existing record, we don't have to lock.
485       */
486      if (whence == SEEK_END) /* we're appending, lock entire file */
487          if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
488              err_dump("_db_writedat: writew_lock error");

489      if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
490          err_dump("_db_writedat: lseek error");
491      db->datlen = strlen(data) + 1;  /* datlen includes newline */

492      iov[0].iov_base = (char *) data;
493      iov[0].iov_len  = db->datlen - 1;
494      iov[1].iov_base = &newline;
495      iov[1].iov_len  = 1;
496      if (writev(db->datfd, &iov[0], 2) != db->datlen)
497          err_dump("_db_writedat: writev error of data record");

498      if (whence == SEEK_END)
499          if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
500              err_dump("_db_writedat: un_lock error");
501    }
```

[472-491] 调用函数_db_writedat写一个数据记录。当删除一条记录时，调用函数_db_writedat清空数据记录，此时_db_writedat并不对数据文件加写锁，因为db_delete对该记录的散列链已经加了写锁，这便保证不再会有其他进程能够读写这条记录。在本节稍后处解说db_store函数时，会遇到_db_writedat函数追加数据文件的情况，此时就必须对该文件加锁。

确定要写数据记录的位置。要写的字节数是记录长度+1字节，这1个字节是为表示记录终止的换行符而增加的。

[492-501] 设置iovec数组，调用writev写数据记录和换行符。因为不能想当然地认为调用者缓冲区的尾端有空间可以加换行符，所以先将换行符送入另一个缓冲，然后再从该缓冲写至数据记录。如果正对文件添加一条记录，则释放早先获得的锁。

737

```
502   /*
503    * Write an index record.  _db_writedat is called before
504    * this function to set the datoff and datlen fields in the
505    * DB structure, which we need to write the index record.
506    */
507   static void
508   _db_writeidx(DB *db, const char *key,
509                off_t offset, int whence, off_t ptrval)
510   {
511     struct iovec   iov[2];
512     char           asciiptrlen[PTR_SZ + IDXLEN_SZ +1];
513     int            len;
514     char           *fmt;

515     if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
516         err_quit("_db_writeidx: invalid ptr: %d", ptrval);
517     if (sizeof(off_t) == sizeof(long long))
518         fmt = "%s%c%lld%c%d\n";
519     else
520         fmt = "%s%c%ld%c%d\n";
521     sprintf(db->idxbuf, fmt, key, SEP, db->datoff, SEP, db->datlen);
522     if ((len = strlen(db->idxbuf)) < IDXLEN_MIN || len > IDXLEN_MAX)
523         err_dump("_db_writeidx: invalid length");
524     sprintf(asciiptrlen, "%*ld%*d", PTR_SZ, ptrval, IDXLEN_SZ, len);

525     /*
526      * If we're appending, we have to lock before doing the lseek
527      * and write to make the two an atomic operation.  If we're
528      * overwriting an existing record, we don't have to lock.
529      */
530     if (whence == SEEK_END)     /* we're appending */
531         if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
532           SEEK_SET, 0) < 0)
533             err_dump("_db_writeidx: writew_lock error");
```

[502-524] 调用_db_writeidx函数写一条索引记录。在验证散列链中下一个指针有效后，创建索引记录，并将它的后半部分存放到idxbuf中。需要索引记录这一部分的长度以创建该记录的前半部分，而前半部分被存放到局部变量asciiptrlen中。注意，基于off_t数据类型的长度，选择传送给sprintf的格式字符串。即使32位的系统也能提供64位的文件偏移量，所以不能假定off_t数据类型的长度。

[525-533] 和_db_writedat一样，只有在向索引文件添加新索引记录时这一函数才需要加锁。_db_dodelete调用此函数是为了重写一条现有的索引记录，在这种情况下调用者已经在散列链上加了写锁，所以不再需要加另外的锁。

738

```
534     /*
535      * Position the index file and record the offset.
536      */
537     if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
538         err_dump("_db_writeidx: lseek error");

539     iov[0].iov_base = asciiptrlen;
540     iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
541     iov[1].iov_base = db->idxbuf;
542     iov[1].iov_len  = len;
543     if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
544         err_dump("_db_writeidx: writev error of index record");
```

```
545     if (whence == SEEK_END)
546         if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
547           SEEK_SET, 0) < 0)
548             err_dump("_db_writeidx: un_lock error");
549 }

550 /*
551  * Write a chain ptr field somewhere in the index file:
552  * the free list, the hash table, or in an index record.
553  */
554 static void
555 _db_writeptr(DB *db, off_t offset, off_t ptrval)
556 {
557     char    asciiptr[PTR_SZ + 1];

558     if (ptrval < 0 || ptrval > PTR_MAX)
559         err_quit("_db_writeptr: invalid ptr: %d", ptrval);
560     sprintf(asciiptr, "%*ld", PTR_SZ, ptrval);

561     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
562         err_dump("_db_writeptr: lseek error to ptr field");
563     if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
564         err_dump("_db_writeptr: write error of ptr field");
565 }
```

[534-549] 设置索引文件偏移量，从此处开始写索引记录，将该偏移量存入DB结构的idxoff字段。因为是在两个分开的缓冲中构造索引记录，所以调用writev将它存放到索引文件中。如果是追加该文件，则释放在定位操作前加的锁。从并发运行的进程添加新记录至同一数据库角度思考问题，那么这把锁使定位(seek)和写成为原子操作。

[550-565] _db_writeptr用于将一个链表指针写至索引文件中。验证该指针在索引文件的边界范围内，然后将它变换成ASCII字符串。按指定的偏移量在索引文件中定位，接着将该指针ASCII字符串写入索引文件。

739

```
566 /*
567  * Store a record in the database.  Return 0 if OK, 1 if record
568  * exists and DB_INSERT specified, -1 on error.
569  */
570 int
571 db_store(DBHANDLE h, const char *key, const char *data, int flag)
572 {
573     DB      *db = h;
574     int     rc, keylen, datlen;
575     off_t   ptrval;

576     if (flag != DB_INSERT && flag != DB_REPLACE &&
577       flag != DB_STORE) {
578         errno = EINVAL;
579         return(-1);
580     }
581     keylen = strlen(key);
582     datlen = strlen(data) + 1;      /* +1 for newline at end */
583     if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
584         err_dump("db_store: invalid data length");

585     /*
586      * _db_find_and_lock calculates which hash table this new record
587      * goes into (db->chainoff), regardless of whether it already
```

```
588	 * exists or not. The following calls to _db_writeptr change the
589	 * hash table entry for this chain to point to the new record.
590	 * The new record is added to the front of the hash chain.
591	 */
592	if (_db_find_and_lock(db, key, 1) < 0) { /* record not found */
593	    if (flag == DB_REPLACE) {
594	        rc = -1;
595	        db->cnt_storerr++;
596	        errno = ENOENT;      /* error, record does not exist */
597	        goto doreturn;
598	    }
```

[566-584] db_store函数用于将一条记录加到数据库中。首先验证参数flag的值；然后，查明数据记录长度是否有效，如果无效，则构造core文件并退出。作为一个例子这样处理无可厚非，但如果构造的是可正式应用的函数库，那么最好返回出错状态而非退出，这样可以给应用程序一个恢复机会。

[585-598] 调用_db_find_and_lock以查看这个记录是否已经存在。如果记录并不存在且指定的标志为DB_INSERT或DB_STORE，或者记录存在且指定的标志为DB_REPLACE或DB_STORE，那么这些都是允许的。替换一条已存在的记录，指的是该记录的键一样，而数据记录很可能不一样。注意，因为db_store很可能会修改散列链，所以用_db_find_and_lock的最后一个参数指明要对散列链加写锁。

740

```
599	    /*
600	     * _db_find_and_lock locked the hash chain for us; read
601	     * the chain ptr to the first index record on hash chain.
602	     */
603	    ptrval = _db_readptr(db, db->chainoff);

604	    if (_db_findfree(db, keylen, datlen) < 0) {
605	        /*
606	         * Can't find an empty record big enough. Append the
607	         * new record to the ends of the index and data files.
608	         */
609	        _db_writedat(db, data, 0, SEEK_END);
610	        _db_writeidx(db, key, 0, SEEK_END, ptrval);

611	        /*
612	         * db->idxoff was set by _db_writeidx.  The new
613	         * record goes to the front of the hash chain.
614	         */
615	        _db_writeptr(db, db->chainoff, db->idxoff);
616	        db->cnt_stor1++;
617	    } else {
618	        /*
619	         * Reuse an empty record. _db_findfree removed it from
620	         * the free list and set both db->datoff and db->idxoff.
621	         * Reused record goes to the front of the hash chain.
622	         */
623	        _db_writedat(db, data, db->datoff, SEEK_SET);
624	        _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
625	        _db_writeptr(db, db->chainoff, db->idxoff);
626	        db->cnt_stor2++;
627	    }
```

[599-603] 在调用_db_find_and_lock后，程序分成四种情况。前两种情况中，没有找到相应的记录，所以添加新记录。读散列链上第一项的偏移量。

[604-616] 第一种情况：调用_db_findfree在空闲链表中搜索一条已删除的记录，它的键长度和数据长度与参数keylen和datlen相同。如果没有找到对应大小的记录。这意味着要将这条新记录添加到索引文件和数据文件的末尾。调用_db_writedat写数据部分，调用_db_writeidx写索引部分，调用_db_writeptr将新记录加到对应的散列链的链首。将对此种情况的执行进行计数的计数器(cnt_stor1)值加1，以便观察数据库的运行状况。

[617-627] 第二种情况：_db_findfree找到了对应大小的空记录，并将这条空记录从空闲链表上移下来（很快就会见到_db_findfree的实现），写入新的索引记录和数据记录，然后，与第一种情况一样，将新记录加到对应的散列链的链首。将对此种情况的执行进行计数的计数器（cnt_stor2）值加1，以便观察数据库的运行状况。

741

```
628     } else {                          /* record found */
629         if (flag == DB_INSERT) {
630             rc = 1;     /* error, record already in db */
631             db->cnt_storerr++;
632             goto doreturn;
633         }

634         /*
635          * We are replacing an existing record.  We know the new
636          * key equals the existing key, but we need to check if
637          * the data records are the same size.
638          */
639         if (datlen != db->datlen) {
640             _db_dodelete(db);   /* delete the existing record */

641             /*
642              * Reread the chain ptr in the hash table
643              * (it may change with the deletion).
644              */
645             ptrval = _db_readptr(db, db->chainoff);

646             /*
647              * Append new index and data records to end of files.
648              */
649             _db_writedat(db, data, 0, SEEK_END);
650             _db_writeidx(db, key, 0, SEEK_END, ptrval);

651             /*
652              * New record goes to the front of the hash chain.
653              */
654             _db_writeptr(db, db->chainoff, db->idxoff);
655             db->cnt_stor3++;
656         } else {
```

[628-633] 另两种情况是具相同键的记录在数据库中已存在。如果不想替换该记录，则设置表示一条记录已经存在的返回码，将对存储出错计数的计数器（cnt_storerr）值加1，然后跳转至函数末尾，在此处理公共返回逻辑。

[634-656] 第三种情况：要替换一条现存记录，而新数据记录的长度与已存在记录的长度不一样。调用_db_dodelete将老记录删除，该删除记录将放在空闲链表的链

首。然后，调用_db_writedat和_db_writeidx将新记录添加到索引文件和数据文件的末尾（也可以用其他方法，如可以再找一找是否有数据大小正好的已删除记录项）。最后调用_db_writeptr将新记录加到对应的散列链的链首。DB结构中的cnt_stor3计数器记录发生此种情况的次数。

742

```
657             /*
658              * Same size data, just replace data record.
659              */
660             _db_writedat(db, data, db->datoff, SEEK_SET);
661             db->cnt_stor4++;
662         }
663     }
664     rc = 0;     /* OK */

665 doreturn: /* unlock hash chain locked by _db_find_and_lock */
666     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
667         err_dump("db_store: un_lock error");
668     return(rc);
669 }

670 /*
671  * Try to find a free index record and accompanying data record
672  * of the correct sizes.  We're only called by db_store.
673  */
674 static int
675 _db_findfree(DB *db, int keylen, int datlen)
676 {
677     int     rc;
678     off_t   offset, nextoffset, saveoffset;

679     /*
680      * Lock the free list.
681      */
682     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
683         err_dump("_db_findfree: writew_lock error");

684     /*
685      * Read the free list pointer.
686      */
687     saveoffset = FREE_OFF;
688     offset = _db_readptr(db, saveoffset);
```

[657-663]　第四种情况：要替换一条现存记录，而新数据记录的长度与已存在的记录的长度恰好一样。这是最容易的情况，只需要重写数据记录即可，并将计数器(cnt_stor4)的值加1。

[664-669]　在正常情况下，设置表示成功的返回码，然后进入公共返回逻辑。对散列链解锁（这把锁是由调用_db_find_and_lock而加上的），然后返回调用者。

[670-688]　_db_findfree函数试图找到一个指定大小的空闲索引记录和相关联的数据记录。_db_findfree需要对空闲链表加写锁以避免与其他使用空闲链表的进程互相干扰。在对空闲链表加写锁后，得到空闲链表链首的指针地址。

743

```
689     while (offset != 0) {
690         nextoffset = _db_readidx(db, offset);
691         if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
692             break;      /* found a match */
```

```
693         saveoffset = offset;
694         offset = nextoffset;
695     }

696     if (offset == 0) {
697         rc = -1;    /* no match found */
698     } else {
699         /*
700          * Found a free record with matching sizes.
701          * The index record was read in by _db_readidx above,
702          * which sets db->ptrval.  Also, saveoffset points to
703          * the chain ptr that pointed to this empty record on
704          * the free list.  We set this chain ptr to db->ptrval,
705          * which removes the empty record from the free list.
706          */
707         _db_writeptr(db, saveoffset, db->ptrval);
708         rc = 0;

709         /*
710          * Notice also that _db_readidx set both db->idxoff
711          * and db->datoff.  This is used by the caller, db_store,
712          * to write the new index record and data record.
713          */
714     }

715     /*
716      * Unlock the free list.
717      */
718     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
719         err_dump("_db_findfree: un_lock error");
720     return(rc);
721 }
```

[689-695] _db_findfree中的while循环遍历空闲链表，以搜寻一个有匹配键长度和数据长度的记录项。在这个简单的实现中，只有当一个已删除记录的键长度及数据长度与要加入的新记录的键长度及数据长度一样时，才重用已删除记录的空间。其他更好的重用删除空间的方法一般更复杂。

[696-714] 如果找不到具有所要求的键长度和数据长度的可用记录，则设置表示失败的返回码。否则，将已找到记录的下一个链表指针值写至前一记录的链表指针，这样就从空闲链表中移除了该记录。

[715-721] 一旦结束对空闲链表的操作，就释放写锁，然后对调用者返回状态码。 744

```
722 /*
723  * Rewind the index file for db_nextrec.
724  * Automatically called by db_open.
725  * Must be called before first db_nextrec.
726  */
727 void
728 db_rewind(DBHANDLE h)
729 {
730     DB      *db = h;
731     off_t   offset;

732     offset = (db->nhash + 1) * PTR_SZ;  /* +1 for free list ptr */

733     /*
734      * We're just setting the file offset for this process
```

```
735      * to the start of the index records; no need to lock.
736      * +1 below for newline at end of hash table.
737      */
738     if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
739         err_dump("db_rewind: lseek error");
740 }
741 /*
742  * Return the next sequential record.
743  * We just step our way through the index file, ignoring deleted
744  * records.  db_rewind must be called before this function is
745  * called the first time.
746  */
747 char *
748 db_nextrec(DBHANDLE h, char *key)
749 {
750     DB      *db = h;
751     char    c;
752     char    *ptr;
```

[722-740] db_rewind函数用于把数据库重置到“起始状态”，将索引文件的文件偏移量定位在索引文件的第一条索引记录（紧跟在散列表之后）。（回忆图20-1中索引文件的结构。）

[741-752] db_nextrec函数返回数据库的下一条记录，返回值是指向数据缓冲的指针。如果调用者提供的Key参数非空，那么相应键复制到该缓冲中。调用者负责分配可以存放键的足够大的缓冲。大小为IDXLEN_MAX字节的缓冲足够存放任一键。记录按它们在数据库文件中存放的顺序逐一返回，因此，记录并不按键值的大小排序。另外，db_nextrec并不跟随散列链表，所以也可能读到已删除的记录，只是不向调用者返回这种已删除记录。

745

```
753     /*
754      * We read lock the free list so that we don't read
755      * a record in the middle of its being deleted.
756      */
757     if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
758         err_dump("db_nextrec: readw_lock error");
759     do {
760         /*
761          * Read next sequential index record.
762          */
763         if (_db_readidx(db, 0) < 0) {
764             ptr = NULL;     /* end of index file, EOF */
765             goto doreturn;
766         }
767         /*
768          * Check if key is all blank (empty record).
769          */
770         ptr = db->idxbuf;
771         while ((c = *ptr++) != 0  &&  c == SPACE)
772             ;   /* skip until null byte or nonblank */
773     } while (c == 0);   /* loop until a nonblank key is found */
774     if (key != NULL)
775         strcpy(key, db->idxbuf);    /* return key */
```

```
776     ptr = _db_readdat(db);  /* return pointer to data buffer */
777     db->cnt_nextrec++;

778   doreturn:
779     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
780         err_dump("db_nextrec: un_lock error");
781     return(ptr);
782   }
```

[753-758] 首先，需要对空闲链表加读锁，使得正在读该链表时，其他进程不能从中删除某一记录。

[759-773] 调用_db_readidx读下一条记录。传送给该函数的偏移量参数值为0，以此通知该函数从当前位置继续读索引记录。因为是逐条顺序地读索引文件，所以可能会读到已删除的记录。仅需返回有效记录，所以跳过键是全空格的记录（回忆_db_dodelete函数以设置为全空格方式清除键）。

[774-782] 当找到一个有效键时，如果调用者已提供缓冲，则将该键复制至该缓冲。然后读数据记录，并将返回值设置为指向包含数据记录的内部缓冲的指针值。使统计计数器值加1，对空闲链表解锁，最后返回指向数据记录的指针。 746

通常在下列形式的循环中的使用db_rewind和db_nextrec这两个函数：

```
db_rewind(db);
while ((ptr = db_nextrec(db, key)) != NULL) {
    /* process record */
}
```

前面曾警告过，记录的返回没有一定的次序，它们并不按键的顺序返回。

如果db_nextrec函数在循环中被调用时数据库正被修改，则db_nextrec返回的记录只是变化中的数据库在某一时间点的快照（snapshot）。db_nextrec被调用时总是返回一条“正确”的记录，也就是说它不会返回一条已删除的记录；但有可能一条记录刚被db_nextrec返回后就被删除；类似地，如果db_nextrec刚跳过一条已删除的记录，这条记录的空间就被一条新记录重用，此时除非用db_rewind并重新遍历一遍，否则将看不到这条新的记录。如果通过db_nextrec获得一份数据库的准确的“冻结”的快照很重要，则应做到在这段时间内没有添加和删除。

下面来看db_nextrec使用的加锁。因为并不使用任何的散列链表，也不能判断每条记录属于哪条散列链，所以有可能当db_nextrec读取一条记录时，其索引记录正在被删除。为了防止这种情况，db_nextrec对空闲链表加读锁，这样就可避免与_db_dodelete和_db_findfree相互影响。

在结束对db.c源文件的解释说明之前，还需要说明在向文件的末尾添加索引记录或数据记录时，需要加锁。在第1情况和第3种情况中，db_store调用_db_writeidx和_db_writedat时，第3个参数为0，第4个参数为SEEK_END。这里，第4个参数作为一个标志用来告诉这两个函数，新的记录将被添加到文件的末尾。_db_writeidx用到的技术是对索引文件加写锁，加锁的范围从散列链的末尾到文件的末尾。这不会影响其他数据库的读用户和写用户（这些用户将对散列链加锁），但如果其他用户此时调用db_store来添加数据则会被阻止。_db_writedat使用的方法是对整个数据文件加写锁。同样这也不会影响其他数据库的读用户和写用户（它们甚至不对数据文件加锁），但如果其他用户此时调用db_store来向数据文件添加数据则会被阻止（见习题20.3）。

20.9 性能

为了测试这一数据库函数库，也为了获得一些与典型应用的数据访问模式有关的时间测试数据，我们编写了一个测试程序。该程序接受两个命令行参数：要创建的子进程的个数以及每个子进程向数据库写的数据库记录的条数（*nrec*），然后创建一个空的数据库（通过调用db_open），通过forks创建指定数目的子进程，并等待所有子进程结束。每个子进程执行以下步骤：

747

(1) 向数据库写*nrec*条记录。

(2) 通过键值读回*nrec*条记录。

(3) 执行下面的循环$nrec \times 5$次：

(a) 随机读一条记录。

(b) 每循环37次，随机删除一条记录。

(c) 每循环11次，添加一条新记录并读回这条记录。

(d) 每循环17次，随机替换一条记录为新记录。在连续两次替换中，一次用同样大小的记录替换，一次用比以前更长的记录替换。

(4) 将此子进程写的所有记录删除。每删除一条记录，随机地寻找10条记录。

随着函数的调用次数增加，DB结构的cnt_xxx变量记录对数据库进行的操作数。每个子进程的操作数一般都会与其他子进程不一样，因为每个子进程用来选择记录的随机数生成器是根据其进程ID来初始化的。当*nrec*为500时，每个子进程的较典型的操作计数见表20-2。

表20-2 *nrec*为500时，每个子进程执行的操作的典型计数

操　作	计　数
db_store，DB_INSERT，无空白记录，添加	678
db_store，DB_INSERT，重用空白记录	164
db_store，DB_REPLACE，数据长度不同，添加	97
db_store，DB_REPLACE，数据长度相同	109
db_store，没有找到记录	19
db_fetch，找到记录	8 114
db_fetch，没有找到记录	732
db_delete，找到记录	842
db_delete，没有找到记录	110

读取的次数大约是存储或删除的10倍，这可能是许多数据库应用程序的典型情况。

每一个子进程只对该子进程所写的记录执行这些操作（读取、存储和删除）。由于所有的子进程对同一个数据库进行操作（虽然对不同的记录），所以会使用并发控制。数据库中的记录总条数与子进程数成比例增加。（当只有一个子进程时，一开始有*nrec*条记录写入数据库；当有两个子进程时，一开始有$nrec \times 2$条记录写入数据库，依此类推。）

通过运行测试程序的三个不同版本来比较加粗锁和加细锁提供的并发，并且比较三种不同的加锁方式（不加锁、建议性锁和强制性锁）。第一个版本加细锁，用20.8节中的源代码，曾将此称为细锁版本；第二个版本通过改变加锁调用而使用粗锁，20.6节对此已介绍过；第三个版本将所有加锁调用均去掉，这样可以计算加锁的开销。通过改变数据库文件的权限标志位，还可以使第一个版本和第二个版本（加细锁和加粗锁）使用建议性锁或强制性锁（本节所有的测

748

试中，仅对加细锁的实现测量了采用强制性锁的时间）。

本节所有的测试都在一台运行Solaris 9的SPARC系统上进行。

1. 单进程的结果

表20-3显示了只有一个子进程运行时的结果，*nrec*分别为500、1 000和2 000。

表20-3 单子进程、不同的*nrec*和不同的加锁方法

nrec	不加锁			建议性锁						强制性锁		
				粗锁			细锁			细锁		
	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock
500	0.42	0.89	1.31	0.42	1.17	1.59	0.41	1.04	1.45	0.46	1.49	1.95
1 000	1.51	3.89	5.41	1.64	4.13	5.78	1.63	4.12	5.76	1.73	6.34	8.07
2 000	3.91	10.06	13.98	4.09	10.30	14.39	4.03	10.63	14.66	4.47	16.21	20.70

最后12列显示的是以秒为单位的时间。在所有的情况下，用户CPU时间加上系统CPU时间都近似地等于时钟时间。这一组测试受CPU限制而不是受磁盘操作限制。

中间6列（建议性锁）对加粗锁和加细锁的结果基本一样。这是可以理解的，因为对于单个进程来说加粗锁和加细锁并没有区别。

比较不加锁和加建议性锁，可以看到加锁调用在系统CPU时间上增加了2%~31%。即使这些锁实际上并没有使用过（由于只有一个进程），`fcntl`系统调用仍会有一些时间的开销。另外，注意到用户CPU时间对于四种不同的加锁方案基本上一样，这是因为用户代码基本上是一样的（除了调用`fcntl`的次数有所不同外）。

关于表20-3要注意的最后一点是强制性锁比建议性锁增加了大约43%~54%的系统CPU时间。由于对加强制细锁和加建议细锁的加锁调用次数是一样的，故增加的系统开销来自读和写。

最后的测试是尝试有多个子进程的不加锁的程序。与预想的一样，结果是随机的错误。一般情况包括：加入到数据库中的记录找不到，测试程序异常退出等。几乎每次运行测试程序，就有不同的错误发生。这是典型的竞争状态—多个进程在没有任何加锁的情况下修改同一个文件，错误情况不可预测。

749

2. 多进程的结果

下一组测试主要查看粗锁和细锁的不同。前面说过，由于加细锁时数据库的各个部分被其他进程锁住的时间比加粗锁少，所以凭直觉加细锁应能够提供更好的并发性。表20-4显示了对*nrec*取500、子进程数目从1到12的测试结果。

所有的用户时间、系统时间和时钟时间的单位均为秒，所有这些时间均是父进程与所有子进程的总和。关于这些数据有许多需要考虑。

第8列（标记为“Δ Clock”）是加建议粗锁与加建议细锁的时钟时间的时间差，从该度量中可以看出使用细锁得到了多大的并发度。在运行测试程序的系统上，当并发进程数不多于7时，加粗锁与加细锁的效果大致相当；即使多于7个进程，使用细锁的时间减少也不大（一般少于3%），这不禁让人怀疑使用额外的代码来实现细锁是否值得。

我们希望从粗锁到细锁时钟时间会减少，最后也确实如此，但也曾预期就系统时间而言，对任何进程数，细锁都将保持比粗锁高。这样预想的原因是对细锁调用了更多次的`fcntl`。如果将表20-2中的`fcntl`调用次数加起来，平均对粗锁有21 730次，细锁25 292次（表20-2中的每个操作对于粗锁要调用两次`fcntl`，而对于细锁前三个`db_store`及记录删除（记录找到）需

750 要调用四次fcntl)。基于此，认为由于增加了16%的fcntl调用次数，所以会增加细锁的系统时间。然而，在测试中当进程数超过7后，加细锁的系统时间反而稍有下降，这不免让人迷惑。

表20-4 *nrec*=500时，不同加锁方法的比较

#Proc	建议性锁							强制性锁			
	粗锁			细锁			Δ	细锁			Δ
	User	Sys	Clock	User	Sys	Clock	Clock	User	Sys	Clock	Percent
1	0.41	1.00	1.42	0.41	1.05	1.47	0.05	0.47	1.40	1.87	33
2	1.10	2.81	3.92	1.11	2.80	3.92	0.00	1.15	4.06	5.22	45
3	2.17	5.27	7.44	2.19	5.18	7.37	−0.07	2.31	7.67	9.99	48
4	3.36	8.55	11.91	3.26	8.67	11.94	0.03	3.51	12.69	16.20	46
5	4.72	13.08	17.80	4.99	12.64	17.64	−0.16	4.91	19.21	24.14	52
6	6.45	17.96	24.42	6.83	17.29	24.14	−0.28	7.03	26.59	33.66	54
7	8.46	23.12	31.62	8.67	22.96	31.65	0.03	9.25	35.47	44.74	54
8	10.83	29.68	40.55	11.00	29.39	40.41	−0.14	11.67	45.90	57.63	56
9	13.35	36.81	50.23	13.43	36.28	49.76	−0.47	14.45	58.02	72.49	60
10	16.35	45.28	61.66	16.09	44.10	60.23	−1.43	17.43	70.90	88.37	61
11	18.97	54.24	73.24	19.13	51.70	70.87	−2.37	20.62	84.98	105.69	64
12	22.92	63.54	86.51	22.94	61.28	84.29	−2.22	24.41	101.68	126.20	66

对此作进一步分析，发现减少的原因是：对粗锁而言，保持这种锁的时间会比较长，于是增加了其他进程因这种锁而阻塞的可能性；而对细锁而言，由于加锁的时间较短，于是进程为此而阻塞的机会也就比较少。如果分析运行12个数据库进程的系统行为，将会看到加粗锁时的进程切换次数是加细锁时的3倍。这就意味着在使用细锁时，进程在锁上阻塞的机会要少得多。

最后一列（标记为“Δ percent”）是从加建议细锁到加强制细锁的系统CPU时间的百分比增量。这与在表20-3中看到的强制性锁显著增加（约33%~66%）系统时间是一致的。

由于所有这些测试的用户代码几乎一样（对加建议细锁和强制细锁增加了一些fcntl调用)，预期对每一行的用户CPU时间应基本一样。

表20-4的第一行与表20-3中的*nrec*取500的那一行很相似。这与预期相一致。

图20-4是表20-4中加建议细锁的数据图。图中绘制了进程数从1到12的时钟时间，也绘制了用户CPU时间除以进程数后的每进程用户CPU时间，另外还绘制了系统CPU时间除以进程数后 751 的每进程系统CPU时间。

注意到这两个每进程CPU时间都是线性的，但时钟时间是非线性的。可能的原因是：当进程数增大时，操作系统用于进行进程切换的CPU时间增多。操作系统的开销是使时钟时间增加，但不会影响单个进程的CPU时间。

用户CPU时间随进程数增加的原因可能是因为数据库中有了更多的记录，每一条散列链更长，所以_db_find_and_lock函数平均要运行更长时间来找到一条记录。

20.10 小结

本章详细介绍了一个数据库函数库的设计与实现。考虑到篇幅，使这个函数库尽可能小和简单，但也包括了多进程并发控制需要的对记录加锁的功能。

此外，还使用不同数目的进程，以及不同的加锁方法：不加锁、建议性锁（细锁和粗锁）和强制性锁，研究了这个函数库的性能。可以看到加建议性锁比不加锁在时钟时间上增加了约10%，加强制性锁比建议性锁耗时再增加约33%~66%。

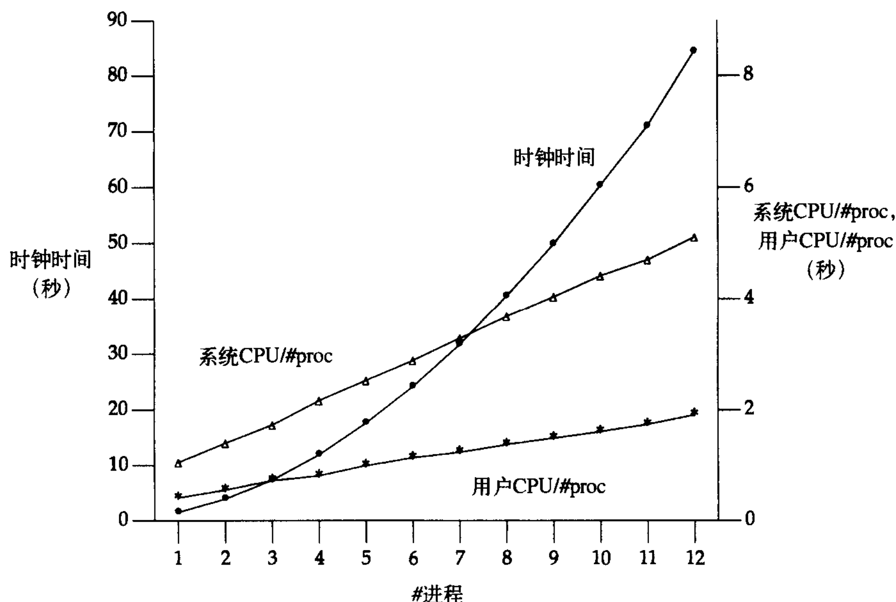


图20-4　表20-4中使用建议细锁的数据

习题

20.1 在`_db_dodelete`中使用的加锁是比较保守的。例如，如果等到真正要用空闲链表时再加锁，则可获得更大的并发度。如果将调用`writew_lock`移到调用`_db_writedat`和`_db_readptr`之间会发生什么呢？

20.2 如果`db_nextrec`不对空闲链表加读锁而它所读的记录正在被删除，描述在怎样的情况下，`db_nextrec`会返回一个正确的键但是数据记录却是空的（提示:查看`_db_dodelete`）。

20.3 在20.8节的结尾部分，描述了`_db_writeidx`和`_db_writedat`的加锁，曾说过这种加锁不会干涉除了调用`db_store`外的其他的读进程和写进程。如果改为强制性锁，这还成立吗？

20.4 怎样把`fsync`集成到这个数据库函数库中？

20.5 在`db_store`中，先写数据记录，然后再写索引记录。如果将次序颠倒，会发生什么？

20.6 建立一个新的数据库并写入一些记录。写一个程序调用`db_nextrec`来读数据库中的每条记录，并调用`_db_hash`来计算每条记录的散列值。根据每条散列链上的记录数画出直方图。`_db_hash`中的散列函数是否适当？

20.7 修改数据库函数，使得索引文件中散列链的数目可以在数据库建立时指定。

20.8 比较两种情况下数据库函数的性能：(a) 数据库与测试程序在同一台机器上；(b) 数据库与测试程序在不同的机器上，经由NFS进行访问。这个数据库函数库提供的记录锁机制还能工作吗？

752

第21章

与网络打印机通信

21.1 引言

现在开发与网络打印机通信的程序，这些打印机通过以太网与多个计算机互联，并且通常既支持纯文本文件又支持PostScript文件。尽管有一些应用程序也支持其他的通信协议，但一般使用网络打印协议（Internet Printing Protocol，IPP）与打印机通信。

我们将描述两个程序：一个打印假脱机守护进程（print spooler daemon），用以将作业发送到打印机；一个命令行程序，用以将打印作业提交到假脱机守护进程。打印假脱机必须处理很多操作（与客户端通信来提交作业，与打印机通信，读文件，扫描目录，等等），这就提供了一个机会来使用前面章节所提到的函数。例如，使用线程（第11章和第12章）来简化打印假脱机程序的设计，使用套接字（第16章）在调度文件打印的程序和打印假脱机程序之间通信，也可以在打印假脱机程序与网络打印机之间通信。

21.2 网络打印协议

网络打印协议IPP为建立基于网络的打印系统指定了通信规则。通过将IPP服务器嵌入到带网卡的打印机中，打印机就能够对许多计算机系统的请求加以服务。这些计算机系统实际上并不需要在同一个物理网络中。因为IPP是建立在标准的因特网协议上的，所以任何一台能够与打印机建立TCP/IP连接的计算机都能向打印机提交打印作业。

特别地，IPP建立在超文本传输协议（Hypertext Transfer Protocol，HTTP）上（见21.3节），HTTP又建立在TCP/IP上。IPP报文的结构如图21-1所示。

Ethernet 首部	IP 首部	TCP 首部	HTTP 首部	IPP 首部	要打印的数据

图21-1 IPP报文结构

IPP是请求响应协议。客户端发送请求到服务器，服务器用响应报文回答这个请求。IPP首部包含一个字段来指示所需操作，这些操作可以定义成提交打印作业、取消打印作业、获取作业属性、获取打印机属性、暂停和重起打印机、挂起一个作业和释放一个挂起的作业。

图21-2显示了IPP报文首部的结构。首部两个字节表示IPP版本号，对于1.1版本协议，每个字节的值是1。对于请求协议，接下来两个字节包含一个值用以指示所请求的操作；对于响应协议，这两个字节包含一个状态码。

接下来的四个字节包含一个整数以标识请求。接着是可选的属性，然后用属性结束标志终止。紧接着属性结束标志之后是任何与请求相关联的数据。

在首部，整数以有符号二进制补码以及大端字节序（即网络字节序）方式存储。属性按照组来存储。每个组都以标识该组的一个字节开始。在每一个组中，属性通常表示为：1字节的标志，然后是2字节的名字长度，接着是属性名字，然后是2字节的属性值长度，最后是属性值本身。属性值可以编码成字符串、二进制整数或者更为复杂的结构，例如日期/时间戳。

754

图21-3显示了attributes-charset属性是如何编码成值utf-8的。

版本号	(2字节)
操作ID（请求）/状态码（响应）	(2字节)
请求ID	(4字节)
属性	(0-*n*字节)
属性结束标志	(1字节)
数据	(0-*n*字节)

图21-2 IPP首部结构

属性标志 = 0×47	(1字节)
属性名长度= 18	(2字节)
属性名= attributes-charset	(18字节)
属性值长度= 5	(2字节)
属性值= utf-8	(5字节)

图21-3 IPP属性编码示例

根据所请求的操作，一些属性需要在请求报文中提供，而另外一些是可选的。例如，表21-1显示了用于打印作业请求的属性。

表21-1 打印作业请求的属性

属　性	状　态	描　述
attributes-charset	必需	text或name类型属性所使用的字符集
attributes-natural-language	必需	text或name类型属性所使用的自然语言
printer-uri	必需	打印机的统一资源标识符（Universal Resource Identifier）
requesting-user-name	可选	提交作业的用户名字（如果可以，用于鉴别）
job-name	可选	用于区别多个作业的作业名字
ipp-attribute-fidelity	可选	如果为真，告诉打印机如果属性不匹配就拒绝作业；否则，打印机尽可能打印作业
document-name	可选	文档名字（例如，适合打印一个旗标）
document-format	可选	文档格式（纯文本、PostScript等）
document-natural-language	可选	文档的自然语言
compression	可选	压缩文档数据的算法
job-k-octets	可选	以1 024字节单位计算的文档大小
job-impressions	可选	作业中提交的图的数量（这里的“图”是嵌入在页面的图像）数量
job-media-sheets	可选	该作业打印张数

IPP首部包含了文本和二进制混合数据。属性名存储为文本，而数据大小存储为二进制整数。这使得构建和分析首部的过程变得复杂，因为需要考虑诸如网络字节序、主机处理器是否能在任意字节边界编址整数之类的问题。一个较好的可选方案是将首部设计成仅仅包含文本。这样以稍微膨胀一些协议报文为代价简化处理过程。

755

IPP由一系列请求评注（Requests For Comments，RFC）文档说明，文档可以从http://www.pwg.org/ipp获得。尽管有许多其他文档来进一步说明管理过程、作业属性等信息，但主要文档在表21-2中列出。

表21-2 主要的IPP RFC

RFC	标 题
2567	Design Goals for an Internet Printing Protocol（IPP设计目标）
2568	Rationale for the Structure of the Model and Protocol for the Internet Printing Protocol（IPP模型与协议架构之基本原理）
2911	Internet Printing Protocol/1.1: Model and Semantics（IPP/1.1：模型与语义）
2910	Internet Printing Protocol/1.1: Encoding and Transport（IPP/1.1：编码与传输）
3196	Internet Printing Protocol/1.1: Implementor's Guide（IPP/1.1：实现者指南）

21.3 超文本传输协议

HTTP V1.1由RFC 2616说明。HTTP也是请求响应协议。请求报文包含一个开始行，跟着是首部行，一个空白行，然后是一个可选的实体主体。在目前这种情况下，实体主体包含IPP首部和数据。

HTTP首部是ASCII码，每行以回车（`\r`）和换行（`\n`）符结束。开始行包含一个*method*来指示客户端请求的操作，一个统一资源定位符（Uniform Resource Locator, URL）来描述服务器和协议，一个字符串来表示HTTP版本。IPP所用的方法仅为`POST`，用于将数据发送到服务器。

首部行指定属性，例如实体主体的格式和长度。一个首部行包含一个属性名字，跟以一个冒号，可选的空格符，然后是属性值，最后以回车和换行符结束。例如，为了指定实体主体包含IPP报文，应包含如下的首部行：

```
Content-Type: application/ipp
```

HTTP响应报文中的开始行包含一个版本字符串，跟着是一个数字状态码和状态消息，以回车和换行符结束。HTTP响应报文的剩余部分格式与请求报文相同：首部之后是一个空白行，然后是一个可选的实体主体。

下面是一个对于作者的打印机的打印请求的HTTP首部样例。

```
POST /phaser860/ipp HTTP/1.1^M
Content-Length: 21931^M
Content-Type: application/ipp^M
Host: phaser860:ipp^M
^M
```

每行后面的`^M`是换行符前的回车符。换行符不能被显示成可打印字符。注意除了回车和换行符，首部的最后一行是空的。

756

21.4 打印假脱机技术

本章中开发的程序是一个简单的打印假脱机程序的基础。一个简单的用户命令发送文件到打印假脱机程序；假脱机程序将其保存到磁盘，把请求送入队列，最终将文件发送到打印机。

所有的UNIX系统至少提供一个打印假脱机系统。FreeBSD安装BSD的打印假脱机系统——LPD（参见`lpd`(8)和Stevens [1990]第13章）。Linux和Mac OS X包含CUPS——Common UNIX Printing System（参见`cupsd`(8)）。Solaris提供标准的SystemV printer spooler（参见`lp`(1)和`lpsched`(1M)）。在本章中，兴趣不在于这些假脱机系统本身，而在于如何与网络打印机通信。需要开发一个假脱机系统能够解决多用户访问单一资源（打印机）的问题。

使用一个简单的命令行程序读取文件，将其发送到打印假脱机守护进程。这个命令行程序有

一个选项用来强制将文件按照纯文本对待（默认是PostScript文件）。这个命令行程序是print。

在打印假脱机守护进程printd中，使用多线程来分担守护进程需要完成的任务。

- 一个线程在套接字上监听从运行print命令的客户端发来的新打印请求。
- 对于每个客户端产生一个独立的线程，用以将要打印的文件复制到假脱机区域。
- 一个线程与打印机通信，一次发送一个队列中的作业。
- 一个线程处理信号。

图21-4显示如何将这些组件整合在一起。

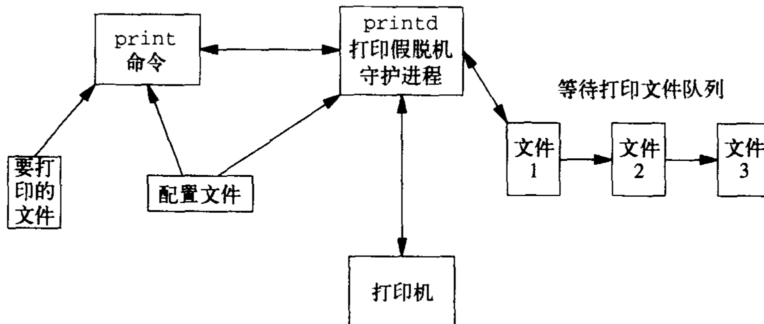


图21-4　打印假脱机组件

打印配置文件是/etc/printer.conf。该文件标识了运行打印假脱机守护进程的服务器主机名字和网络打印机的主机名字。以printserver关键字开始的行标识了假脱机守护进程，关键字之后是空格符和服务器主机名字；以printer关键字开始的行标识了打印机，关键字之后是空格符和打印机的主机名字。

757

一个打印机配置文件样例可能包含下列行：

```
printserver   blade
printer       phaser860
```

其中，blade是运行打印假脱机守护进程的计算机系统主机名字，phaser860是网络打印机的主机名字。

安全

拥有超级用户特权的程序可能让计算机系统受到攻击。这些程序通常并不比其他程序更脆弱，但是被攻破后将导致攻击者能够完全访问计算机系统。

本章中的打印假脱机守护进程拥有超级用户特权，在这个例子中能够将一个特权TCP端口号绑定一个套接字。为使得守护进程更加能够抵御攻击，可以：

- 按照最少特权的原则（8.11节）设计守护进程程序。在获得绑定到特权端口地址的套接字之后，可以将守护进程的用户和组的ID更改为非root（例如lp）。所有用于存储队列中打印作业的文件和目录的拥有者应该是非特权用户。如果被攻击，这种情况下守护进程程序给予攻击者的只有打印子系统。虽然这仍是一个隐患，但是比攻击者可以完全访问系统的危害性大大降低。
- 审查守护进程程序源代码中已知的所有脆弱漏洞，比如缓冲区溢出。
- 对不期望或者可疑的行为做日志，这样管理员可以对此引起注意并作进一步调查。

21.5 源代码

本章的源代码有五个文件，不包括在前面章节中所用的一些公共库例程：

ipp.h 包含IPP定义的头文件

print.h 包含常用的常数、数据结构定义以及工具例程声明的头文件

util.c 用于两个程序的工具例程

print.c 用于打印文件的命令行程序C源文件

printd.c 打印假脱机守护进程的C源文件

接下来按照所列次序依次学习每个文件。从ipp.h头文件开始。 758

```
 1  #ifndef _IPP_H
 2  #define _IPP_H
 3  /*
 4   * Defines parts of the IPP protocol between the scheduler
 5   * and the printer.  Based on RFC2911 and RFC2910.
 6   */
 7  /*
 8   * Status code classes.
 9   */
10  #define STATCLASS_OK(x)     ((x) >= 0x0000 && (x) <= 0x00ff)
11  #define STATCLASS_INFO(x)   ((x) >= 0x0100 && (x) <= 0x01ff)
12  #define STATCLASS_REDIR(x)  ((x) >= 0x0200 && (x) <= 0x02ff)
13  #define STATCLASS_CLIERR(x) ((x) >= 0x0400 && (x) <= 0x04ff)
14  #define STATCLASS_SRVERR(x) ((x) >= 0x0500 && (x) <= 0x05ff)
15  /*
16   * Status codes.
17   */
18  #define STAT_OK             0x0000  /* success */
19  #define STAT_OK_ATTRIGN     0x0001  /* OK; some attrs ignored */
20  #define STAT_OK_ATTRCON     0x0002  /* OK; some attrs conflicted */
21  #define STAT_CLI_BADREQ     0x0400  /* invalid client request */
22  #define STAT_CLI_FORBID     0x0401  /* request is forbidden */
23  #define STAT_CLI_NOAUTH     0x0402  /* authentication required */
24  #define STAT_CLI_NOPERM     0x0403  /* client not authorized */
25  #define STAT_CLI_NOTPOS     0x0404  /* request not possible */
26  #define STAT_CLI_TIMOUT     0x0405  /* client too slow */
27  #define STAT_CLI_NOTFND     0x0406  /* no object found for URI */
28  #define STAT_CLI_OBJGONE    0x0407  /* object no longer available */
29  #define STAT_CLI_TOOBIG     0x0408  /* requested entity too big */
30  #define STAT_CLI_TOOLNG     0x0409  /* attribute value too large */
31  #define STAT_CLI_BADFMT     0x040a  /* unsupported doc format */
32  #define STAT_CLI_NOTSUP     0x040b  /* attributes not supported */
33  #define STAT_CLI_NOSCHM     0x040c  /* URI scheme not supported */
34  #define STAT_CLI_NOCHAR     0x040d  /* charset not supported */
35  #define STAT_CLI_ATTRCON    0x040e  /* attributes conflicted */
36  #define STAT_CLI_NOCOMP     0x040f  /* compression not supported */
37  #define STAT_CLI_COMPERR    0x0410  /* data can't be decompressed */
38  #define STAT_CLI_FMTERR     0x0411  /* document format error */
39  #define STAT_CLI_ACCERR     0x0412  /* error accessing data */
```

[1-14] ipp.h以标准的#ifdef开始，用于防止同一文件被包含两次的错误。然后定义IPP状态码的类（参见RFC2911的13小节）

[15-39] 定义基于RFC 2911的状态码，但是本程序中并不使用，这些状态码的使用留给

759

读者作为练习（参见习题21.1）。

```
40  #define STAT_SRV_INTERN   0x0500  /* unexpected internal error */
41  #define STAT_SRV_NOTSUP   0x0501  /* operation not supported */
42  #define STAT_SRV_UNAVAIL  0x0502  /* service unavailable */
43  #define STAT_SRV_BADVER   0x0503  /* version not supported */
44  #define STAT_SRV_DEVERR   0x0504  /* device error */
45  #define STAT_SRV_TMPERR   0x0505  /* temporary error */
46  #define STAT_SRV_REJECT   0x0506  /* server not accepting jobs */
47  #define STAT_SRV_TOOBUSY  0x0507  /* server too busy */
48  #define STAT_SRV_CANCEL   0x0508  /* job has been canceled */
49  #define STAT_SRV_NOMULTI  0x0509  /* multi-doc jobs unsupported */

50  /*
51   * Operation IDs
52   */
53  #define OP_PRINT_JOB           0x02
54  #define OP_PRINT_URI           0x03
55  #define OP_VALIDATE_JOB        0x04
56  #define OP_CREATE_JOB          0x05
57  #define OP_SEND_DOC            0x06
58  #define OP_SEND_URI            0x07
59  #define OP_CANCEL_JOB          0x08
60  #define OP_GET_JOB_ATTR        0x09
61  #define OP_GET_JOBS            0x0a
62  #define OP_GET_PRINTER_ATTR    0x0b
63  #define OP_HOLD_JOB            0x0c
64  #define OP_RELEASE_JOB         0x0d
65  #define OP_RESTART_JOB         0x0e
66  #define OP_PAUSE_PRINTER       0x10
67  #define OP_RESUME_PRINTER      0x11
68  #define OP_PURGE_JOBS          0x12

69  /*
70   * Attribute Tags.
71   */
72  #define TAG_OPERATION_ATTR     0x01  /* operation attributes tag */
73  #define TAG_JOB_ATTR           0x02  /* job attributes tag */
74  #define TAG_END_OF_ATTR        0x03  /* end of attributes tag */
75  #define TAG_PRINTER_ATTR       0x04  /* printer attributes tag */
76  #define TAG_UNSUPP_ATTR        0x05  /* unsupported attributes tag */
```

[40-49]　继续定义状态码。0x500到0x5ff是服务器错误码。RFC 2911中13.1.1节到13.1.5节描述了所有的状态码。

[50-68]　接着定义各种操作ID。IPP中定义的每个操作（参见RFC 2911的4.4.15节）都有一个ID。在本例中，仅用到打印作业操作。

[69-76]　属性标志限定了IPP请求报文和响应报文中的属性组。这些标志值定义在RFC 2910的3.5.1节中。

760

```
77  /*
78   * Value Tags.
79   */
80  #define TAG_UNSUPPORTED        0x10 /* unsupported value */
81  #define TAG_UNKNOWN            0x12 /* unknown value */
82  #define TAG_NONE               0x13 /* no value */
83  #define TAG_INTEGER            0x21 /* integer */
84  #define TAG_BOOLEAN            0x22 /* boolean */
```

```
85   #define TAG_ENUM             0x23 /* enumeration */
86   #define TAG_OCTSTR           0x30 /* octetString */
87   #define TAG_DATETIME         0x31 /* dateTime */
88   #define TAG_RESOLUTION       0x32 /* resolution */
89   #define TAG_INTRANGE         0x33 /* rangeOfInteger */
90   #define TAG_TEXTWLANG        0x35 /* textWithLanguage */
91   #define TAG_NAMEWLANG        0x36 /* nameWithLanguage */
92   #define TAG_TEXTWOLANG       0x41 /* textWithoutLanguage */
93   #define TAG_NAMEWOLANG       0x42 /* nameWithoutLanguage */
94   #define TAG_KEYWORD          0x44 /* keyword */
95   #define TAG_URI              0x45 /* URI */
96   #define TAG_URISCHEME        0x46 /* uriScheme */
97   #define TAG_CHARSET          0x47 /* charset */
98   #define TAG_NATULANG         0x48 /* naturalLanguage */
99   #define TAG_MIMETYPE         0x49 /* mimeMediaType */

100  struct ipp_hdr {
101    int8_t  major_version;  /* always 1 */
102    int8_t  minor_version;  /* always 1 */
103    union {
104        int16_t op; /* operation ID */
105        int16_t st; /* status */
106    } u;
107    int32_t request_id;     /* request ID */
108    char    attr_group[1];  /* start of optional attributes group */
109    /* optional data follows */
110  };

111  #define operation u.op
112  #define status u.st

113  #endif /* _IPP_H */
```

[77-99] 值标志指示每个属性和参数的格式，由RFC 2910的3.5.2节定义。

[100-113] 定义IPP首部的结构。请求报文与响应报文的首部一样，除了请求中的操作ID被响应中的状态码代替。

在头文件尾部用`#endif`来匹配文件开始处的`#ifdef`。

761

下一个文件是`print.h`头文件。

```
1   #ifndef _PRINT_H
2   #define _PRINT_H
3   /*
4    * Print server header file.
5    */
6   #include <sys/socket.h>
7   #include <arpa/inet.h>
8   #if defined(BSD) || defined(MACOS)
9   #include <netinet/in.h>
10  #endif
11  #include <netdb.h>
12  #include <errno.h>

13  #define CONFIG_FILE    "/etc/printer.conf"
14  #define SPOOLDIR       "/var/spool/printer"
15  #define JOBFILE        "jobno"
16  #define DATADIR        "data"
17  #define REQDIR         "reqs"

18  #define FILENMSZ        64
```

```
19  #define FILEPERM          (S_IRUSR|S_IWUSR)
20  #define USERNM_MAX        64
21  #define JOBNM_MAX         256
22  #define MSGLEN_MAX        512

23  #ifndef HOST_NAME_MAX
24  #define HOST_NAME_MAX     256
25  #endif

26  #define IPP_PORT          631
27  #define QLEN              10
28  #define IBUFSZ            512     /* IPP header buffer size */
29  #define HBUFSZ            512     /* HTTP header buffer size */
30  #define IOBUFSZ           8192    /* data buffer size */
```

[1-12]　在这个头文件中包含所需要的所有头文件。应用程序只需简单地包含print.h，而不需要跟踪所有的头文件依赖关系。

[13-17]　定义实现所需的文件和目录。需要打印的文件副本存放在目录/var/spool/printer/data中，对于每个请求的控制信息存储在目录/var/spool/printer/reqs中。包含下一个作业号的文件是/var/spool/printer/jobno。

[18-30]　接着定义限制和常数。FILEPERM是创建提交打印文件副本时使用的权限，该权限很受限，因为不想让一个普通用户能够读取另外一个用户等待打印的文件。IPP定义成使用端口631。QLEN是传给listen的*backlog*参数（详情参见16.4小节）。

762

```
31  #ifndef ETIME
32  #define ETIME ETIMEDOUT
33  #endif

34  extern int getaddrlist(const char *, const char *,
35    struct addrinfo **);
36  extern char *get_printserver(void);
37  extern struct addrinfo *get_printaddr(void);
38  extern ssize_t tread(int, void *, size_t, unsigned int);
39  extern ssize_t treadn(int, void *, size_t, unsigned int);
40  extern int connect_retry(int, const struct sockaddr *, socklen_t);
41  extern int initserver(int, struct sockaddr *, socklen_t, int);

42  /*
43   * Structure describing a print request.
44   */
45  struct printreq {
46     long size;                 /* size in bytes */
47     long flags;                /* see below */
48     char usernm[USERNM_MAX];   /* user's name */
49     char jobnm[JOBNM_MAX];     /* job's name */
50  };

51  /*
52   * Request flags.
53   */
54  #define PR_TEXT        0x01    /* treat file as plain text */

55  /*
56   * The response from the spooling daemon to the print command.
57   */
58  struct printresp {
```

```
59      long retcode;                   /* 0=success, !0=error code */
60      long jobid;                     /* job ID */
61      char msg[MSGLEN_MAX];           /* error message */
62  };
63  #endif /* _PRINT_H */
```

[31-33] 由于一些平台不定义错误ETIME，因此另外定义一个错误码，使得在这些系统上有意义。

[34-41] 接着，声明所有包含在util.c中的公共例程（接下来将考查这些例程）。注意程序清单16-2中的connect_retry函数和程序清单16-9中的initserver函数没有包含在util.c中。

[42-63] printreq和printresp结构定义了print命令行程序和打印假脱机守护进程之间的协议。print命令发送printreq结构到打印假脱机守护进程，该结构定义了用户名字、作业名字与文件大小。打印假脱机守护进程用printresp结构回应，该结构包括返回码、作业ID和错误消息（当请求失败时）。 763

接下来将考查文件util.c，该文件包含工具例程。

```
 1  #include "apue.h"
 2  #include "print.h"
 3  #include <ctype.h>
 4  #include <sys/select.h>

 5  #define MAXCFGLINE 512
 6  #define MAXKWLEN   16
 7  #define MAXFMTLEN  16

 8  /*
 9   * Get the address list for the given host and service and
10   * return through ailistpp.  Returns 0 on success or an error
11   * code on failure.  Note that we do not set errno if we
12   * encounter an error.
13   *
14   * LOCKING: none.
15   */
16  int
17  getaddrlist(const char *host, const char *service,
18    struct addrinfo **ailistpp)
19  {
20      int             err;
21      struct addrinfo hint;

22      hint.ai_flags = AI_CANONNAME;
23      hint.ai_family = AF_INET;
24      hint.ai_socktype = SOCK_STREAM;
25      hint.ai_protocol = 0;
26      hint.ai_addrlen = 0;
27      hint.ai_canonname = NULL;
28      hint.ai_addr = NULL;
29      hint.ai_next = NULL;
30      err = getaddrinfo(host, service, &hint, ailistpp);
31      return(err);
32  }
```

[1-7] 首先定义了这个文件中的函数所需的限制。MAXCFGLINE是打印机配置文件中

行的最大长度，MAXKWLEN是配置文件中关键字的最大尺寸，MAXFMTLEN是传给sscanf的格式化字符串的最大长度。

[8-32] 第一个函数是getaddrlist，它是getaddrinfo（16.3.3节）的封装，因为常常用同样的结构来调用getaddrinfo。注意在这个函数中不需要用互斥锁。每个函数前面的LOCKING注释只为用于多线程锁定的文档编写。这一注释列出了可能有的关于锁的假设，告知该函数所需要获得或释放的锁，并告知调用者这个函数所需要持有的锁。

764

```
33  /*
34   * Given a keyword, scan the configuration file for a match
35   * and return the string value corresponding to the keyword.
36   *
37   * LOCKING: none.
38   */
39  static char *
40  scan_configfile(char *keyword)
41  {
42      int             n, match;
43      FILE            *fp;
44      char            keybuf[MAXKWLEN], pattern[MAXFMTLEN];
45      char            line[MAXCFGLINE];
46      static char     valbuf[MAXCFGLINE];

47      if ((fp = fopen(CONFIG_FILE, "r")) == NULL)
48          log_sys("can't open %s", CONFIG_FILE);
49      sprintf(pattern, "%%%ds %%%ds", MAXKWLEN-1, MAXCFGLINE-1);
50      match = 0;
51      while (fgets(line, MAXCFGLINE, fp) != NULL) {
52          n = sscanf(line, pattern, keybuf, valbuf);
53          if (n == 2 && strcmp(keyword, keybuf) == 0) {
54              match = 1;
55              break;
56          }
57      }
58      fclose(fp);
59      if (match != 0)
60          return(valbuf);
61      else
62          return(NULL);
63  }
```

[33-46] scan_configfile函数在打印机配置文件中搜索指定的关键字。

[47-63] 以读方式打开配置文件，根据搜索模式建立格式字符串。符号%%%ds建立一个格式指示器来限定字符串尺寸，这样就不会溢出用于在堆栈中存放字符串的缓冲区。在文件中一次读取一行，搜索用空格符分开的两个字符串；如果找到这样的字符串，就用关键字比较第一个字符串。如果找到一个匹配或者读取到文件尾，则循环结束并关闭文件。当关键字匹配时，返回一个指针指向包含关键字后面的字符串的缓冲区；否则，返回NULL。

返回的字符串存放在一个静态缓冲区（valbuf），该缓冲区会被紧接着的调用覆盖。因此，scan_configfile不能用于多线程程序，除非小心地避免同时有多个线程调用它。

765

```
64   /*
65    * Return the host name running the print server or NULL on error.
66    *
67    * LOCKING: none.
68    */
69   char *
70   get_printserver(void)
71   {
72       return(scan_configfile("printserver"));
73   }

74   /*
75    * Return the address of the network printer or NULL on error.
76    *
77    * LOCKING: none.
78    */
79   struct addrinfo *
80   get_printaddr(void)
81   {
82       int             err;
83       char            *p;
84       struct addrinfo *ailist;

85       if ((p = scan_configfile("printer")) != NULL) {
86           if ((err = getaddrlist(p, "ipp", &ailist)) != 0) {
87               log_msg("no address information for %s", p);
88               return(NULL);
89           }
90           return(ailist);
91       }
92       log_msg("no printer address specified");
93       return(NULL);
94   }
```

[64-73] get_printserver仅仅是一个函数封装器，它调用scan_configfile来找到运行打印假脱机守护进程的计算机系统名字。

[74-94] 使用get_printaddr函数来获取网络打印机的地址。该函数和前面的函数很像，除了当找到配置文件中的打印机名字时用名字来查找相应的网络地址。get_printserver和get_printaddr均调用scan_configfile。如果不能打开打印机配置文件，scan_configfile就调用log_sys打印出错信息并退出。尽管get_printserver想要由客户端命令调用，而get_printaddr想要由守护进程程序调用，两者均可调用log_sys，因为通过设置一个全局变量可以安排日志函数将其打印到标准错误，而不是输出到日志文件。 766

```
 95  /*
 96   * "Timed" read - timout specifies the # of seconds to wait before
 97   * giving up (5th argument to select controls how long to wait for
 98   * data to be readable).  Returns # of bytes read or -1 on error.
 99   *
100   * LOCKING: none.
101   */
102  ssize_t
103  tread(int fd, void *buf, size_t nbytes, unsigned int timout)
104  {
105      int             nfds;
106      fd_set          readfds;
```

```
107     struct timeval  tv;

108     tv.tv_sec = timout;
109     tv.tv_usec = 0;
110     FD_ZERO(&readfds);
111     FD_SET(fd, &readfds);
112     nfds = select(fd+1, &readfds, NULL, NULL, &tv);
113     if (nfds <= 0) {
114         if (nfds == 0)
115             errno = ETIME;
116         return(-1);
117     }
118     return(read(fd, buf, nbytes));
119 }
```

[95-107] 我们提供叫作tread的函数来读取指定的字节数，在放弃以前至多阻塞*timout*秒。当我们从一个套接字或一个管道读数据时这个函数很有用。如果在指定的时间期限内没有接受数据，返回−1并将errno设为ETIME。如果在时间期限内有数据可用，返回最多*nbytes*字节的数据，但是如果数据没有及时到达，我们可以返回比要求的要少。

我们将使用tread在打印假脱机守护进程上预防拒绝服务攻击。一个恶意用户可能重复尝试连接到守护进程而不发送数据，不让其他用户能够提交打印作业。通过一个合理时间内放弃的方式，我们防止这种情况发生。其巧妙之处在于选择一个合理的超时值，当系统负载比较低和任务花费更长时间时，该值足够大能够防止过早夭折。但是，如果我们选择的值太大，通过允许守护进程消耗太多资源去处理挂起请求，可能导致拒绝服务攻击。

[108-119] 使用select等待指定的文件描述符可读。如果在要读取的数据可用之前超时，select返回0，这种情况将errno设为ETIME。如果select失败或超时，返回−1；否则，返回任何可用数据。

767

```
120 /*
121  * "Timed" read - timout specifies the number of seconds to wait
122  * per read call before giving up, but read exactly nbytes bytes.
123  * Returns number of bytes read or -1 on error.
124  *
125  * LOCKING: none.
126  */
127 ssize_t
128 treadn(int fd, void *buf, size_t nbytes, unsigned int timout)
129 {
130     size_t  nleft;
131     ssize_t nread;

132     nleft = nbytes;
133     while (nleft > 0) {
134         if ((nread = tread(fd, buf, nleft, timout)) < 0) {
135             if (nleft == nbytes)
136                 return(-1); /* error, return -1 */
137             else
138                 break;      /* error, return amount read so far */
139         } else if (nread == 0) {
140             break;          /* EOF */
141         }
```

```
142             nleft -= nread;
143             buf += nread;
144         }
145         return(nbytes - nleft);       /* return >= 0 */
146     }
```

[120-146] 还提供了tread的变体，叫做treadn，它只正好读取请求的字节数。这和14.8节中描述的readn函数类似，只是增加了一个超时参数。

为了正好读取*nbytes*字节，必须准备进行多次read调用。其困难之处在于尝试将单个超时值应用到多个read调用。这里不想用闹钟，因为在多线程应用中信号会变乱；也不能依赖系统根据select的返回的更新过的timeval结构，以指示剩余的时间，因为许多平台不支持这个（14.5.1节）。因此，这种情况需要折中并定义一个超时值应用到单独的read调用。代替限制等待时间的总量，它限制循环中每次迭代的等待时间。总等待的最大时间由（*nbytes*×*timout*）秒限定（最坏的情况，一次仅收到一个字节）。

使用*nleft*来记录要读取的剩余字节数。如果tread失败并在上一次迭代中已经接收到数据，则停止while循环并返回读取的字节数；否则返回-1。

768

接下来是用于提交打印作业的命令程序，它的C源文件是print.c。

```
 1  /*
 2   * The client command for printing documents.  Opens the file
 3   * and sends it to the printer spooling daemon.  Usage:
 4   *     print [-t] filename
 5   */
 6  #include "apue.h"
 7  #include "print.h"
 8  #include <fcntl.h>
 9  #include <pwd.h>

10  /*
11   * Needed for logging funtions.
12   */
13  int log_to_stderr = 1;

14  void submit_file(int, int, const char *, size_t, int);

15  int
16  main(int argc, char *argv[])
17  {
18      int             fd, sockfd, err, text, c;
19      struct stat     sbuf;
20      char            *host;
21      struct addrinfo *ailist, *aip;

22      err = 0;
23      text = 0;
24      while ((c = getopt(argc, argv, "t")) != -1) {
25          switch (c) {
26          case 't':
27              text = 1;
28              break;

29          case '?':
30              err = 1;
```

```
31                  break;
32              }
33          }
```

[1-14] 需要定义一个名为log_to_stderr的整数，通过这个整数能够使用函数库中的日志函数。如果该整数设为非零值，错误消息将被送到标准错误输出来取代日志文件。尽管在print.c中没有使用任何日志函数，但确实将util.o链接到print.o来构建可执行的print命令，并且util.c包含用于用户命令行程序和守护进程的函数。

[15-33] 支持一个选项：-t，强迫文件按照文本格式打印（而不是其他格式，例如PostScript格式）。使用getopt(3)函数来处理命令选项。

769

```
34      if (err || (optind != argc - 1))
35          err_quit("usage: print [-t] filename");
36      if ((fd = open(argv[optind], O_RDONLY)) < 0)
37          err_sys("print: can't open %s", argv[1]);
38      if (fstat(fd, &sbuf) < 0)
39          err_sys("print: can't stat %s", argv[1]);
40      if (!S_ISREG(sbuf.st_mode))
41          err_quit("print: %s must be a regular file\n", argv[1]);

42      /*
43       * Get the hostname of the host acting as the print server.
44       */
45      if ((host = get_printserver()) == NULL)
46          err_quit("print: no print server defined");
47      if ((err = getaddrlist(host, "print", &ailist)) != 0)
48          err_quit("print: getaddrinfo error: %s", gai_strerror(err));

49      for (aip = ailist; aip != NULL; aip = aip->ai_next) {
50          if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
51              err = errno;
52          } else if (connect_retry(sockfd, aip->ai_addr,
53            aip->ai_addrlen) < 0) {
54              err = errno;
```

[34-41] 当getopt处理完命令选项时，将变量optind设为指向第一个非选项参数下标。如果该值不是最后一个参数的下标，那么指定了错误的参数个数（只支持一个非选项参数）。错误处理包括：检查是否能够打开要打印的文件；检查是否是一个常规文件（而不是一个目录或者其他类型的文件）。

[42-48] 通过调用util.c中的get_printserver函数，取得打印假脱机守护进程的名字，然后调用getaddrlist（也在util.c中）将主机名字转换成网络地址。注意：指定服务名字为“print”。在系统上安装打印假脱机守护进程时，需要确保/etc/services（或等价的数据库）有打印机服务的条目。当为守护进程程序选择端口时，最好选择特权端口，以防止恶意用户程序假装成一个打印假脱机守护进程，而实际上是要偷取打印文件的副本。这意味着端口号应小于1 024（回忆16.3.4节），并且守护进程程序运行时必须具有超级用户特权以便能够绑定一个保留端口。

[49-54] 从由getaddrinfo返回的地址列表中，一次使用一个地址来尝试连接到守护进程程序，然后使用能够连接的第一个地址来发送文件到守护进程程序。

770

```
55          } else {
56              submit_file(fd, sockfd, argv[1], sbuf.st_size, text);
57              exit(0);
58          }
59      }
60      errno = err;
61      err_ret("print: can't contact %s", host);
62      exit(1);
63  }

64  /*
65   * Send a file to the printer daemon.
66   */
67  void
68  submit_file(int fd, int sockfd, const char *fname, size_t nbytes,
69              int text)
70  {
71      int                 nr, nw, len;
72      struct passwd       *pwd;
73      struct printreq     req;
74      struct printresp    res;
75      char                buf[IOBUFSZ];

76      /*
77       * First build the header.
78       */
79      if ((pwd = getpwuid(geteuid())) == NULL)
80          strcpy(req.usernm, "unknown");
81      else
82          strcpy(req.usernm, pwd->pw_name);
83      req.size = htonl(nbytes);
84      if (text)
85          req.flags = htonl(PR_TEXT);
86      else
87          req.flags = 0;
```

[55-63] 如果能够建立一个连接，调用submit_file将文件传送到打印假脱机守护进程；如果不能连接到任何地址，则打印错误消息并退出。这里使用err_ret和exit代替单一调用err_sys以避免编译警告，因为如果单独调用err_sys，main的最后一行就不是return语句或exit调用了，从而产生编译警告。

[64-87] submit_file发送打印请求到守护进程程序并读取响应消息。首先，建立printreq请求首部。使用geteuid来获得调用者的有效用户ID并将其传给getpwuid以便在系统密码文件中查找用户，复制这个用户名到请求首部；或者，如果不能标识用户，则使用字符串“unknown”。将要打印的文件转成网络字节序后，把其文件长度保存在请求首部。如果文件按纯文本格式打印，在请求首部保存PR_TEXT标志。

771

```
88      if ((len = strlen(fname)) >= JOBNM_MAX) {
89          /*
90           * Truncate the filename (+-5 accounts for the leading
91           * four characters and the terminating null).
92           */
93          strcpy(req.jobnm, "... ");
94          strncat(req.jobnm, &fname[len-JOBNM_MAX+5], JOBNM_MAX-5);
95      } else {
```

```
 96            strcpy(req.jobnm, fname);
 97        }

 98        /*
 99         * Send the header to the server.
100         */
101        nw = writen(sockfd, &req, sizeof(struct printreq));
102        if (nw != sizeof(struct printreq)) {
103            if (nw < 0)
104                err_sys("can't write to print server");
105            else
106                err_quit("short write (%d/%d) to print server",
107                  nw, sizeof(struct printreq));
108        }

109        /*
110         * Now send the file.
111         */
112        while ((nr = read(fd, buf, IOBUFSZ)) != 0) {
113            nw = writen(sockfd, buf, nr);
114            if (nw != nr) {
115                if (nw < 0)
116                    err_sys("can't write to print server");
117                else
118                    err_quit("short write (%d/%d) to print server",
119                      nw, nr);
120            }
121        }
```

[88-108] 将作业名设为要打印的文件名，如果名字长度超出了报文所能容纳的长度，则将名字的开头部分截去，并代入省略符，以表示该字段还有更多的字符。然后使用writen将请求首部发送到守护进程程序，如果写入失败或者传送的数据少于期望的数据，将打印错误消息然后退出。

[109-121] 在发送请求首部给守护进程程序之后，发送要打印的文件。一次读取文件中IOBUFSZ字节然后使用writen将数据发送给守护进程程序。和请求首部一样，如果写入失败或者写入的数据少于期望的数据，将打印错误消息然后退出。

772

```
122        /*
123         * Read the response.
124         */
125        if ((nr = readn(sockfd, &res, sizeof(struct printresp))) !=
126          sizeof(struct printresp))
127            err_sys("can't read response from server");
128        if (res.retcode != 0) {
129            printf("rejected: %s\n", res.msg);
130            exit(1);
131        } else {
132            printf("job ID %ld\n", ntohl(res.jobid));
133        }
134        exit(0);
135    }
```

[122-135] 当把要打印的文件送给守护进程后，读取守护进程的响应。如果请求失败，返回码（retcode）为非零值，并且将响应中的文本形式的错误信息打印出来。如果请求成功，我们打印作业ID，以便用户以后以此引用该请求。（将写一个命

令来取消一个打印请求作为一个练习；作业ID可以用于取消作业请求，其作用是从打印队列中识别要删除的作业。)

注意，一个成功的从守护进程来的响应并不意味着打印机可以打印该文件，仅仅意味着守护进程成功地将其加入到打印作业队列。

print.c中的大部分内容已经在前面的章节中讨论过了。唯一没有涉及的主题是getopt函数，尽管在第19章的pty程序中已经见到过该函数。

同一个系统中的所有命令遵循同样的使用惯例是很重要的，因为这样更易于使用。如果用户比较熟悉一个命令程序中命令行选项的安排，在操作遵循其他使用惯例的命令时，比较容易出错。

当处理命令行中的空格时这种问题就会浮现。一些命令要求选项与其参数用空格分开，但是另外一些命令要求参数紧随其选项之后，中间没有任何空白。如果没有一套统一的规则，用户要么必须记住所有命令的语法，要么就是不得不在调用命令时反复试验、不断摸索。

Single UNIX Specification包含一套惯例和准则来给出一致的命令行语法，里面包括的建议如“将每个命令行选项限制为一个字母数字字符”和“所有的选项必须以-字符为前导”。

幸运的是，存在getopt函数帮助命令开发者按照一致的方式来处理命令行选项。 773

```
#include <fcntl.h>

int getopt(int argc, const * const argv[], const char *options);

extern int optind, opterr, optopt;
extern char *optarg;
```

返回值：下一个选项字符，若全部选项处理完毕则返回-1

参数*argc*和*argv*与传给main函数的相同。参数*options*是包含命令支持的选项字符的字符串。如果一个选项字符后面跟着一个冒号，那么选项带一个参数；否则，只有选项本身。例如，如果一个命令的用法如下：

```
command [-i] [-u username] [-z] filename
```

则“iu:z”将作为*options*字符串传入到getopt。

getopt的通常用法是一个循环，当getopt返回-1时结束循环。在每次循环中，getopt会返回下一个处理的选项。由应用程序来处理选项中的冲突，实际上，getopt仅仅简单地分析命令选项并强制执行选项的标准格式而已。

当遇到一个不合法的选项时，getopt返回一个问号而不是一个字符。如果选项的参数缺失，getopt也会返回一个问号，但是如果选项字符串中第一个字符是冒号，getopt会返回一个冒号。特殊模式--会让getopt停止处理并返回-1。这可以让用户提供一个以负号开头的命令参数而非选项。例如，如果有一个文件名为-bar，则不可以通过输入下列命令来删除之。

```
rm -bar
```

因为rm会尝试将-bar解释成选项。可以用下列方式来删除文件：

```
rm -- -bar
```

getopt函数支持四个外部变量：

optarg　如果选项带有参数，处理该选项时，getopt将optarg指向该选项的参数字符串。

opterr　如果遇到一个选项错误，getopt默认动作是会打印一个错误消息。为了禁止该行为，应用程序可以将opterr设为0。

optind　下一个要处理的字符串的argv数组的下标。该下标从1开始，每个参数被getopt处理后增量。

774

optopt　如果在处理选项过程中遇到错误，getopt将optopt指向引起错误的选项字符串。

最后一个要考查的C源文件是打印假脱机守护进程程序。

```
 1  /*
 2   * Print server daemon.
 3   */
 4  #include "apue.h"
 5  #include "print.h"
 6  #include "ipp.h"
 7  #include <fcntl.h>
 8  #include <dirent.h>
 9  #include <ctype.h>
10  #include <pwd.h>
11  #include <pthread.h>
12  #include <strings.h>
13  #include <sys/select.h>
14  #include <sys/uio.h>

15  /*
16   * These are for the HTTP response from the printer.
17   */
18  #define HTTP_INFO(x)    ((x) >= 100 && (x) <= 199)
19  #define HTTP_SUCCESS(x) ((x) >= 200 && (x) <= 299)

20  /*
21   * Describes a print job.
22   */
23  struct job {
24      struct job      *next;      /* next in list */
25      struct job      *prev;      /* previous in list */
26      long             jobid;     /* job ID */
27      struct printreq  req;       /* copy of print request */
28  };

29  /*
30   * Describes a thread processing a client request.
31   */
32  struct worker_thread {
33      struct worker_thread  *next;    /* next in list */
34      struct worker_thread  *prev;    /* previous in list */
35      pthread_t              tid;     /* thread ID */
36      int                    sockfd;  /* socket */
37  };
```

[1-19]　打印假脱机守护进程程序包含前面看到的IPP头文件，因为守护进程程序需要和采用这个协议的打印机通信。HTTP_INFO和HTTP_SUCCESS宏定义了HTTP请求的状态（IPP建立在HTTP之上）。

[20-37]　假脱机守护进程程序使用job和worker_thread结构来跟踪相应的打印作业和接受打印请求的线程。

775

```
38  /*
39   * Needed for logging.
40   */
41  int                     log_to_stderr = 0;
```

```
42   /*
43    * Printer-related stuff.
44    */
45   struct addrinfo         *printer;
46   char                    *printer_name;
47   pthread_mutex_t         configlock = PTHREAD_MUTEX_INITIALIZER;
48   int                     reread;

49   /*
50    * Thread-related stuff.
51    */
52   struct worker_thread    *workers;
53   pthread_mutex_t         workerlock = PTHREAD_MUTEX_INITIALIZER;
54   sigset_t                mask;

55   /*
56    * Job-related stuff.
57    */
58   struct job              *jobhead, *jobtail;
59   int                     jobfd;
```

[38-41] 日志函数需要定义log_to_stderr变量，并且将其设为0，以使得日志消息发送到系统日志而不是标准错误输出。在print.c中，即使在用户命令中不使用日志函数，也设置log_to_stderr为1。如果将工具函数拆分为两个单独的文件：一个用于服务器，另一个用于客户端命令，则可以避免这种情况。

[42-48] 使用全局变量printer来保存打印机的网络地址。在printer_name中保存打印机的主机名字。configlock互斥量用于保护对reread变量的访问，该变量用于表示守护进程程序需要再次读取配置文件，原因可能是管理员改变了打印机或打印机的网络地址。

[49-54] 接下来定义与线程相关的变量。使用workers作为线程的双向链表的头部，该表用于接受来自客户端的文件。采用workerlock互斥量来保护该表。变量mask用于线程的信号掩码。

[55-59] 对于挂起作业的链表，定义jobhead为表头，jobtail为表尾。该表也是双向链接的，但是需要将作业加入到表尾，所以需要一个指针来记住表尾。至于表中的工作者线程，次序则是无关紧要的，因此可以将他们加入到表头部而不需要记住表尾指针。jobfd是作业文件的文件描述符。

776

```
60   long                    nextjob;
61   pthread_mutex_t         joblock = PTHREAD_MUTEX_INITIALIZER;
62   pthread_cond_t          jobwait = PTHREAD_COND_INITIALIZER;

63   /*
64    * Function prototypes.
65    */
66   void        init_request(void);
67   void        init_printer(void);
68   void        update_jobno(void);
69   long        get_newjobno(void);
70   void        add_job(struct printreq *, long);
71   void        replace_job(struct job *);
72   void        remove_job(struct job *);
73   void        build_qonstart(void);
74   void        *client_thread(void *);
75   void        *printer_thread(void *);
```

```
76  void        *signal_thread(void *);
77  ssize_t     readmore(int, char **, int, int *);
78  int         printer_status(int, struct job *);
79  void        add_worker(pthread_t, int);
80  void        kill_workers(void);
81  void        client_cleanup(void *);

82  /*
83   * Main print server thread.  Accepts connect requests from
84   * clients and spawns additional threads to service requests.
85   *
86   * LOCKING: none.
87   */
88  int
89  main(int argc, char *argv[])
90  {
91      pthread_t           tid;
92      struct addrinfo     *ailist, *aip;
93      int                 sockfd, err, i, n, maxfd;
94      char                *host;
95      fd_set              rendezvous, rset;
96      struct sigaction    sa;
97      struct passwd       *pwdp;
```

[60-62] nextjob是接收的下一个打印作业的ID。互斥量joblock保护作业链表，同时还有条件变量jobwait代表的条件。

[63-81] 声明本文件中所有余下的函数的原型。把这些工作做在前面可以使得在文件中放置函数时不用担心函数调用的次序。

[82-97] 打印假脱机守护进程程序的main函数执行两个任务：初始化守护进程程序然后处理来自客户端的连接请求。

777

```
 98     if (argc != 1)
 99         err_quit("usage: printd");
100     daemonize("printd");

101     sigemptyset(&sa.sa_mask);
102     sa.sa_flags = 0;
103     sa.sa_handler = SIG_IGN;
104     if (sigaction(SIGPIPE, &sa, NULL) < 0)
105         log_sys("sigaction failed");
106     sigemptyset(&mask);
107     sigaddset(&mask, SIGHUP);
108     sigaddset(&mask, SIGTERM);
109     if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
110         log_sys("pthread_sigmask failed");
111     init_request();
112     init_printer();

113 #ifdef _SC_HOST_NAME_MAX
114     n = sysconf(_SC_HOST_NAME_MAX);
115     if (n < 0)  /* best guess */
116 #endif
117         n = HOST_NAME_MAX;

118     if ((host = malloc(n)) == NULL)
119         log_sys("malloc error");
120     if (gethostname(host, n) < 0)
121         log_sys("gethostname error");
```

[98-100] 守护进程程序没有任何选项，所以如果argc不为1，调用err_quit打印错误消息然后退出。调用程序清单13-1中的daemonize函数成为一个守护进程。在此之后，不能在标准错误上打印错误消息，取而代之的是对其记录日志。

[101-112] 调整为忽略SIGPIPE。将要写套接字文件描述符，并且不想让写错误触发SIGPIPE，因为其默认动作是杀死进程。下一步，设置线程的信号掩码，包括SIGHUP和SIGTERM。创建的所有线程均继承这个信号掩码。使用SIGHUP来告诉守护进程程序再次读取配置文件，SIGTERM来告诉守护进程程序执行清理工作并优雅地退出。调用init_request初始化作业请求并确保只有一个守护进程的副本在运行，调用init_printer来初始化打印机信息（马上可以看到这些函数）。

[113-121] 如果平台定义了_SC_HOST_NAME_MAX符号，调用sysconf来获取主机名字的最大长度。如果sysconf失败或者没有定义该限制，就采用HOST_NAME_MAX作为最佳选择。有时，平台已对此作了定义，但是如果没有，则在print.h中选择值。分配内存来保存主机名字并调用gethostname来获取。

778

```
122      if ((err = getaddrlist(host, "print", &ailist)) != 0) {
123          log_quit("getaddrinfo error: %s", gai_strerror(err));
124          exit(1);
125      }
126      FD_ZERO(&rendezvous);
127      maxfd = -1;
128      for (aip = ailist; aip != NULL; aip = aip->ai_next) {
129          if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
130            aip->ai_addrlen, QLEN)) >= 0) {
131              FD_SET(sockfd, &rendezvous);
132              if (sockfd > maxfd)
133                  maxfd = sockfd;
134          }
135      }
136      if (maxfd == -1)
137          log_quit("service not enabled");

138      pwdp = getpwnam("lp");
139      if (pwdp == NULL)
140          log_sys("can't find user lp");
141      if (pwdp->pw_uid == 0)
142          log_quit("user lp is privileged");
143      if (setuid(pwdp->pw_uid) < 0)
144          log_sys("can't change IDs to user lp");
```

[122-135] 接下来，尝试找到守护进程程序用以提供打印假脱机服务的网络地址。清零rendezvous fd_set变量，该变量将与select一起用来等待客户端连接请求。将最大文件描述符初始化为-1，以确保所分配的第一个文件描述符大于maxfd。对于每个需要提供服务的网络地址，调用initserver（见程序清单16-9）来分配和初始化套接字。如果initserver成功，将其文件描述符加入fd_set，如果该描述符大于现有最大值maxfd，将maxfd设为该描述符值。

[136-137] 当走完整个addrinfo结构列表后，如果maxfd仍为-1，不能启动打印假脱机服务，记录日志然后退出。

[138-144] 守护进程程序需要超级用户特权来绑定套接字到保留端口。完成绑定后，通过

将用户ID改变到用户lp（回忆21.4节的安全方面的讨论）降低该程序特权。这里想遵循最小特权原则，以避免在守护进程程序中将系统暴露给任何可能的攻击。调用getpwnam来找到与用户lp相关的密码条目。如果没有此用户，或者lp具有超级用户特权，记录日志然后退出，否则，调用setuid将实际和有效用户ID改为lp用户ID。为了避免暴露系统，如果不能减小特权，那么就选择不提供任何服务。

779

```
145     pthread_create(&tid, NULL, printer_thread, NULL);
146     pthread_create(&tid, NULL, signal_thread, NULL);
147     build_qonstart();

148     log_msg("daemon initialized");

149     for (;;) {
150         rset = rendezvous;
151         if (select(maxfd+1, &rset, NULL, NULL, NULL) < 0)
152             log_sys("select failed");
153         for (i = 0; i <= maxfd; i++) {
154             if (FD_ISSET(i, &rset)) {

155                 /*
156                  * Accept the connection and handle
157                  * the request.
158                  */
159                 sockfd = accept(i, NULL, NULL);
160                 if (sockfd < 0)
161                     log_ret("accept failed");
162                 pthread_create(&tid, NULL, client_thread,
163                   (void *)sockfd);
164             }
165         }
166     }
167     exit(1);
168 }
```

[145-148] 调用pthread_create两次，创建一个处理信号的线程和一个与打印机通信的线程。(通过限制打印机只与一个线程通信，可以简化与打印机相关的数据结构的锁定。）然后调用build_qonstart在/var/spool/printer目录中搜索任何挂起的作业。对于找到的每个作业，将建立一个结构，让打印机线程知道要将该作业的文件送到打印机。到此，完成守护进程程序的设置，因此记录日志，表明守护进程程序初始化成功完成。

[149-168] 将rendezvous fd_set结构复制到rset，然后调用select等待其中的一个文件描述符变为可读。必须复制rendezvous，因为select会修改传入的fd_set结构来包含满足事件的文件描述符。既然服务器已经将套接字初始化完毕，一个可读的文件描述符就意味着一个连接请求需要处理。当select返回时，检查rset来获取可读的文件描述符。如果找到一个，调用accept接受该连接请求。如果失败，记录日志然后继续检查更多的可读文件描述符；否则，创建一个线程来处理客户端请求。主线程main一直循环，将请求发送到其他线程处理，永远不应到达exit语句。

780

```
169  /*
170   * Initialize the job ID file.  Use a record lock to prevent
171   * more than one printer daemon from running at a time.
172   *
173   * LOCKING: none, except for record-lock on job ID file.
174   */
175  void
176  init_request(void)
177  {
178      int     n;
179      char    name[FILENMSZ];

180      sprintf(name, "%s/%s", SPOOLDIR, JOBFILE);
181      jobfd = open(name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
182      if (write_lock(jobfd, 0, SEEK_SET, 0) < 0)
183          log_quit("daemon already running");

184      /*
185       * Reuse the name buffer for the job counter.
186       */
187      if ((n = read(jobfd, name, FILENMSZ)) < 0)
188          log_sys("can't read job file");
189      if (n == 0)
190          nextjob = 1;
191      else
192          nextjob = atol(name);
193  }
```

[169-183] 函数init_request做两件事情：在作业文件/var/spool/printer/jobno上放一个记录锁，然后读该文件以确定下一个要赋予的作业号。在整个文件上放置一把写锁，表明守护进程程序正在运行。如果当前已有一个守护进程程序正在运行，想启动另外的打印假脱机守护进程程序，这些额外的程序将无法获得写锁，并将退出。因此，同时只能有一个守护进程程序在运行。（在程序清单13-2中使用过这种技术；在14.3节中讨论过write_lock宏。）

[184-193] 作业文件包含一个ASCII码整数字符串，该字符串表示下一个作业号。如果文件刚创建并因此为空，那么将nextjob设为1；否则，使用atol把字符串转换为整数并将其作为下一个作业号。让jobfd对于作业文件保持打开状态，这样当作业创建时就能够更新作业号。不能关闭该文件，因为这将释放已经放置在上面的写锁。

在一个长整型为64位的系统上，至少需要一个21字节的缓冲区来容纳代表最大长整型数的字符串。这里重用文件名字的缓冲区，因为在print.h中FILENMSZ定义为64。

781

```
194  /*
195   * Initialize printer information.
196   *
197   * LOCKING: none.
198   */
199  void
200  init_printer(void)
201  {
202      printer = get_printaddr();
203      if (printer == NULL) {
```

```
204            log_msg("no printer device registered");
205            exit(1);
206        }
207        printer_name = printer->ai_canonname;
208        if (printer_name == NULL)
209            printer_name = "printer";
210        log_msg("printer is %s", printer_name);
211    }

212    /*
213     * Update the job ID file with the next job number.
214     *
215     * LOCKING: none.
216     */
217    void
218    update_jobno(void)
219    {
220        char    buf[32];

221        lseek(jobfd, 0, SEEK_SET);
222        sprintf(buf, "%ld", nextjob);
223        if (write(jobfd, buf, strlen(buf)) < 0)
224            log_sys("can't update job file");
225    }
```

[194-211] init_printer函数用于设置打印机名字和地址。调用get_printaddr（来自util.c）获得打印机地址，如果失败，记录日志并退出。但是不能使用log_sys记录日志，因为get_printaddr可能没有设置errno就失败了。实际上，当其失败并设置errno后，get_printaddr自己会对错误信息记录日志。将打印机名字设为addrinfo结构中的ai_canonname，如果该字段为空，将打印机名字设为默认值printer。注意，将使用的打印机名字记录在日志中，以帮助管理员能够诊断问题。

[212-225] update_jobno函数用于在作业文件/var/spool/printer/jobno中写入下一个作业号。首先，找到文件开头。然后，将整数作业号转换为字符串并写入文件。如果写入失败，记录日志并退出。

782

```
226    /*
227     * Get the next job number.
228     *
229     * LOCKING: acquires and releases joblock.
230     */
231    long
232    get_newjobno(void)
233    {
234        long    jobid;

235        pthread_mutex_lock(&joblock);
236        jobid = nextjob++;
237        if (nextjob <= 0)
238            nextjob = 1;
239        pthread_mutex_unlock(&joblock);
240        return(jobid);
241    }

242    /*
```

```
243     * Add a new job to the list of pending jobs.  Then signal
244     * the printer thread that a job is pending.
245     *
246     * LOCKING: acquires and releases joblock.
247     */
248    void
249    add_job(struct printreq *reqp, long jobid)
250    {
251      struct job  *jp;

252      if ((jp = malloc(sizeof(struct job))) == NULL)
253          log_sys("malloc failed");
254      memcpy(&jp->req, reqp, sizeof(struct printreq));
```

[226-241] get_newjobno函数用于获得下一个作业号。首先将joblock互斥量锁住，对nextjob变量增量，当超出范围时回绕。然后对互斥量解锁并返回增量前的nextjob值。多个线程可以同时调用get_newjobno，需要串行化访问下一个作业号，以使得每个线程得到一个唯一的作业号。（参考图11-4，考察如果在这种情况下不串行化线程会发生什么情况。）

[242-254] add_job函数用于在挂起的打印作业列表末尾增加一个新的打印请求。首先为job结构分配空间，如果失败，记录日志并退出。此时，打印请求已经安全地存在磁盘上，当打印假脱机守护进程程序重起时，会重新读取这些请求。当为新作业分配空间完毕后，将客户端的请求结构复制到作业结构中。在print.h中作业结构job包含一对指针列表、一个作业ID和一个从客户端print命令发送过来的printreq结构副本。

783

```
255      jp->jobid = jobid;
256      jp->next = NULL;
257      pthread_mutex_lock(&joblock);
258      jp->prev = jobtail;
259      if (jobtail == NULL)
260          jobhead = jp;
261      else
262          jobtail->next = jp;
263      jobtail = jp;
264      pthread_mutex_unlock(&joblock);
265      pthread_cond_signal(&jobwait);
266    }

267    /*
268     * Replace a job back on the head of the list.
269     *
270     * LOCKING: acquires and releases joblock.
271     */
272    void
273    replace_job(struct job *jp)
274    {
275      pthread_mutex_lock(&joblock);
276      jp->prev = NULL;
277      jp->next = jobhead;
278      if (jobhead == NULL)
279          jobtail = jp;
280      else
281          jobhead->prev = jp;
```

```
282     jobhead = jp;
283     pthread_mutex_unlock(&joblock);
284 }
```

[255-266]　将作业ID保存并锁住joblock互斥量以获得对打印作业链表的独占访问。将在该列表末尾增加新的作业结构。将新的作业结构的向前指针（previous pointer）指向列表中最后一个作业。如果列表为空，将jobhead指向新的结构；否则，将列表中最后一项的向后指针（next pointer）指向新的结构。然后设置jobtail指向新的结构。对互斥量解锁，然后给打印机线程发信号，告诉该线程另一个作业可用了。

[267-284]　函数replace_job用于将作业插入到挂起作业列表头部。需要获得joblock互斥量，将job结构中的向前指针（previous pointer）设为空（null），将向后指针（next pointer）指向表头。如果列表为空，将jobtail指向插入的job结构；否则，将列表中第一个job结构的向前指针指向插入的job结构。然后将jobhead指针指向插入的job结构，成为新的表头。最后，释放joblock互斥量。

784

```
285 /*
286  * Remove a job from the list of pending jobs.
287  *
288  * LOCKING: caller must hold joblock.
289  */
290 void
291 remove_job(struct job *target)
292 {
293     if (target->next != NULL)
294         target->next->prev = target->prev;
295     else
296         jobtail = target->prev;
297     if (target->prev != NULL)
298         target->prev->next = target->next;
299     else
300         jobhead = target->next;
301 }

302 /*
303  * Check the spool directory for pending jobs on start-up.
304  *
305  * LOCKING: none.
306  */
307 void
308 build_qonstart(void)
309 {
310     int             fd, err, nr;
311     long            jobid;
312     DIR             *dirp;
313     struct dirent   *entp;
314     struct printreq req;
315     char            dname[FILENMSZ], fname[FILENMSZ];

316     sprintf(dname, "%s/%s", SPOOLDIR, REQDIR);
317     if ((dirp = opendir(dname)) == NULL)
318         return;
```

[285-301]　给定要删除的作业的指针，remove_job将作业从挂起的作业列表中删除。调

用者首先须持有joblock互斥量。如果向后指针不为空，将下一个条目的向前指针指向被删除目标的向前指针所指向的条目；否则，该条目为列表中最后一个，因此将jobtail指向被删除目标的向前指针所指向的条目。如果被删除目标的向前指针不为空，将前一个条目的向后指针指向被删除目标的向后指针所指向的条目；否则，该条目是列表中第一个条目，将jobhead设为指向被删除目标后面的那个条目。

[302-318] 当守护进程程序启动时，调用build_qonstart从存储在/var/spool/printer/reqs中的磁盘文件建立一个内存中的打印作业列表。如果不能打开该目录，表示没有打印作业要处理，因此就返回。

785

```
319     while ((entp = readdir(dirp)) != NULL) {
320         /*
321          * Skip "." and ".."
322          */
323         if (strcmp(entp->d_name, ".") == 0 ||
324           strcmp(entp->d_name, "..") == 0)
325             continue;
326         /*
327          * Read the request structure.
328          */
329         sprintf(fname, "%s/%s/%s", SPOOLDIR, REQDIR, entp->d_name);
330         if ((fd = open(fname, O_RDONLY)) < 0)
331             continue;
332         nr = read(fd, &req, sizeof(struct printreq));
333         if (nr != sizeof(struct printreq)) {
334             if (nr < 0)
335                 err = errno;
336             else
337                 err = EIO;
338             close(fd);
339             log_msg("build_qonstart: can't read %s: %s",
340               fname, strerror(err));
341             unlink(fname);
342             sprintf(fname, "%s/%s/%s", SPOOLDIR, DATADIR,
343               entp->d_name);
344             unlink(fname);
345             continue;
346         }
347         jobid = atol(entp->d_name);
348         log_msg("adding job %ld to queue", jobid);
349         add_job(&req, jobid);
350     }
351     closedir(dirp);
352 }
```

[319-325] 在目录中一次读取一个条目，忽略.和..。

[326-346] 对于每个条目，创建文件的完全路径名并打开文件以供读取。如果open调用失败，就跳过该文件。否则，读取保存在文件中的printreq结构。如果不能读取整个结构，关闭该文件，记录日志并unlink该文件。那么就建立了相应数据文件的完全路径名，然后unlink该文件。

[347-352] 如果能够读取完整的printreq结构，则将文件名转换为作业ID（文件名就是其作业ID），记录日志，然后将请求加入到挂起的打印作业列表。当读完整个目

786

录后，readdir返回NULL，关闭目录然后返回。

```
353  /*
354   * Accept a print job from a client.
355   *
356   * LOCKING: none.
357   */
358  void *
359  client_thread(void *arg)
360  {
361      int                 n, fd, sockfd, nr, nw, first;
362      long                jobid;
363      pthread_t           tid;
364      struct printreq     req;
365      struct printresp    res;
366      char                name[FILENMSZ];
367      char                buf[IOBUFSZ];

368      tid = pthread_self();
369      pthread_cleanup_push(client_cleanup, (void *)tid);
370      sockfd = (int)arg;
371      add_worker(tid, sockfd);

372      /*
373       * Read the request header.
374       */
375      if ((n = treadn(sockfd, &req, sizeof(struct printreq), 10)) !=
376        sizeof(struct printreq)) {
377          res.jobid = 0;
378          if (n < 0)
379              res.retcode = htonl(errno);
380          else
381              res.retcode = htonl(EIO);
382          strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
383          writen(sockfd, &res, sizeof(struct printresp));
384          pthread_exit((void *)1);
385      }
```

[353-371] 当连接请求被接受时，main线程中派生出client_thread，其作用是从客户端print命令中接收要打印的文件。为每个客户端打印请求分别创建一个独立的线程。

首先是安装线程清理处理程序（参见11.5节中线程清理处理程序的讨论）。清理处理程序是client_cleanup，将在后面用到。它仅带一个参数：线程ID。然后调用add_worker来创建一个worker_thread结构并将其加入到活跃的客户端线程列表中。

[372-385] 此时，完成了线程的初始化任务，因此从客户端读取请求首部。如果客户端发送的数据少于期望或遇到错误，则响应一个消息，该消息指出错误的原因，然后调用pthread_exit结束线程。

787

```
386      req.size = ntohl(req.size);
387      req.flags = ntohl(req.flags);

388      /*
389       * Create the data file.
390       */
391      jobid = get_newjobno();
```

```
392     sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
393     if ((fd = creat(name, FILEPERM)) < 0) {
394         res.jobid = 0;
395         if (n < 0)
396             res.retcode = htonl(errno);
397         else
398             res.retcode = htonl(EIO);
399         log_msg("client_thread: can't create %s: %s", name,
400           strerror(res.retcode));
401         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
402         writen(sockfd, &res, sizeof(struct printresp));
403         pthread_exit((void *)1);
404     }

405     /*
406      * Read the file and store it in the spool directory.
407      */
408     first = 1;
409     while ((nr = tread(sockfd, buf, IOBUFSZ, 20)) > 0) {
410         if (first) {
411             first = 0;
412             if (strncmp(buf, "%!PS", 4) != 0)
413                 req.flags |= PR_TEXT;
414         }
```

[386-404] 将请求首部中的整数字段转换成主机字节序，调用get_newjobno来保存该打印请求的下一个作业ID。建立作业数据文件，文件名为/var/spool/printer/data/*jobid*，其中*jobid*是请求的作业ID。使用权限许可来防止其他人能够读取这些文件（print.h中定义FILEPERM为S_IRUSR|S_IWUSR）。如果不能创建该文件，则记录错误日志，发送失败响应给客户端，调用pthread_exit结束线程。

[405-414] 读取来自客户端的文件内容，要将其写入数据文件的私有副本中。但是在写任何东西之前，需要在第一次循环时检查一下是否为PostScript文件。如果该文件不是以%!PS模式开头，可以假定其为纯文本文件，这种情况下在请求首部中设置PR_TEXT标志。(如果在执行print命令时包含了-t标志，那么客户端也会设置此标志。）尽管PostScript程序不要求以模式%!PS开始，但文档格式指南(Adobe Systems [1999]）强烈推荐这种方式。

788

```
415         nw = write(fd, buf, nr);
416         if (nw != nr) {
417             if (nw < 0)
418                 res.retcode = htonl(errno);
419             else
420                 res.retcode = htonl(EIO);
421             log_msg("client_thread: can't write %s: %s", name,
422               strerror(res.retcode));
423             close(fd);
424             strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
425             writen(sockfd, &res, sizeof(struct printresp));
426             unlink(name);
427             pthread_exit((void *)1);
428         }
429     }
430     close(fd);
```

```
431     /*
432      * Create the control file.
433      */
434     sprintf(name, "%s/%s/%ld", SPOOLDIR, REQDIR, jobid);
435     fd = creat(name, FILEPERM);
436     if (fd < 0) {
437         res.jobid = 0;
438         if (n < 0)
439             res.retcode = htonl(errno);
440         else
441             res.retcode = htonl(EIO);
442         log_msg("client_thread: can't create %s: %s", name,
443           strerror(res.retcode));
444         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
445         writen(sockfd, &res, sizeof(struct printresp));
446         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
447         unlink(name);
448         pthread_exit((void *)1);
449     }
```

[415-430] 将从客户端读取的数据写入到数据文件。如果write失败，记录出错消息日志，关闭数据文件的文件描述符，发送出错消息给客户端，删除数据文件，调用pthread_exit终止线程。注意，我们不需要显式关闭套接字文件描述符。当调用pthread_exit时，线程清理处理程序会处理这些事情。

当接收到所有要打印的数据时，关闭数据文件的文件描述符。

[431-449] 接下来，创建文件/var/spool/printer/reqs/*jobid*以记住打印请求。如果失败，则记录错误日志，发送出错响应给客户端，删除数据文件，并终止线程。

789

```
450     nw = write(fd, &req, sizeof(struct printreq));
451     if (nw != sizeof(struct printreq)) {
452         res.jobid = 0;
453         if (nw < 0)
454             res.retcode = htonl(errno);
455         else
456             res.retcode = htonl(EIO);
457         log_msg("client_thread: can't write %s: %s", name,
458           strerror(res.retcode));
459         close(fd);
460         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
461         writen(sockfd, &res, sizeof(struct printresp));
462         unlink(name);
463         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
464         unlink(name);
465         pthread_exit((void *)1);
466     }
467     close(fd);

468     /*
469      * Send response to client.
470      */
471     res.retcode = 0;
472     res.jobid = htonl(jobid);
473     sprintf(res.msg, "request ID %ld", jobid);
474     writen(sockfd, &res, sizeof(struct printresp));

475     /*
476      * Notify the printer thread, clean up, and exit.
477      */
```

```
478     log_msg("adding job %ld to queue", jobid);
479     add_job(&req, jobid);
480     pthread_cleanup_pop(1);
481     return((void *)0);
482 }
```

[450-466] 将printreq写入控制文件。如果出错，则记录日志，关闭控制文件描述符，发送失败响应给客户端，删除数据和控制文件，并终止线程。

[467-474] 关闭控制文件的文件描述符，并发送消息给客户端，该消息包括作业ID和成功状态（retcode设为0）。

[475-482] 调用add_job将接收到的作业加入到挂起的打印作业列表中，调用pthread_cleanup_pop完成清理过程，当返回时线程终止。

注意，线程退出之前，必须关闭不再使用的任何文件描述符。与进程终止不同，当一个线程结束并且进程中仍有其他线程时，文件描述符不会自动关闭。如果不关闭不需要的文件描述符，终将耗尽资源。

790

```
483 /*
484 * Add a worker to the list of worker threads.
485 *
486 * LOCKING: acquires and releases workerlock.
487 */
488 void
489 add_worker(pthread_t tid, int sockfd)
490 {
491     struct worker_thread    *wtp;

492     if ((wtp = malloc(sizeof(struct worker_thread))) == NULL) {
493         log_ret("add_worker: can't malloc");
494         pthread_exit((void *)1);
495     }
496     wtp->tid = tid;
497     wtp->sockfd = sockfd;
498     pthread_mutex_lock(&workerlock);
499     wtp->prev = NULL;
500     wtp->next = workers;
501     if (workers == NULL)
502         workers = wtp;
503     else
504         workers->prev = wtp;
505     pthread_mutex_unlock(&workerlock);
506 }

507 /*
508  * Cancel (kill) all outstanding workers.
509  *
510  * LOCKING: acquires and releases workerlock.
511  */
512 void
513 kill_workers(void)
514 {
515     struct worker_thread    *wtp;

516     pthread_mutex_lock(&workerlock);
517     for (wtp = workers; wtp != NULL; wtp = wtp->next)
518         pthread_cancel(wtp->tid);
519     pthread_mutex_unlock(&workerlock);
520 }
```

[483-506] add_worker将一个worker_thread结构加入到活动线程列表中。分配该结构需要的内存，初始化它，锁住workerlock互斥量，将结构加入到列表的头部，然后解锁互斥量。

[507-520] kill_workers函数遍历工作者线程列表然后一一删除，遍历列表时持有workerlock。注意，pthread_cancel仅仅安排线程删除，实际的删除动作在每个线程到达下一个删除点时发生。

791

```
521    /*
522     * Cancellation routine for the worker thread.
523     *
524     * LOCKING: acquires and releases workerlock.
525     */
526    void
527    client_cleanup(void *arg)
528    {
529        struct worker_thread    *wtp;
530        pthread_t               tid;

531        tid = (pthread_t)arg;
532        pthread_mutex_lock(&workerlock);
533        for (wtp = workers; wtp != NULL; wtp = wtp->next) {
534            if (wtp->tid == tid) {
535                if (wtp->next != NULL)
536                    wtp->next->prev = wtp->prev;
537                if (wtp->prev != NULL)
538                    wtp->prev->next = wtp->next;
539                else
540                    workers = wtp->next;
541                break;
542            }
543        }
544        pthread_mutex_unlock(&workerlock);
545        if (wtp != NULL) {
546            close(wtp->sockfd);
547            free(wtp);
548        }
549    }
```

[521-543] 函数client_cleanup是与客户端命令通信的工作者线程的线程清理处理程序。当线程调用pthread_exit时，用一个非零参数调用pthread_cleanup_pop时，或者响应一个删除请求时，client_cleanup函数被调用。其参数是终止线程的线程ID。

我们锁住workerlock互斥量然后搜索工作者线程列表，直到找到一个匹配的线程ID。当我们找到一个匹配时，将其从列表中删除工作者线程结构并且停止搜索。

[544-549] 解锁workerlock互斥量，关闭线程用于和客户端通信的套接字文件描述符，然后释放worker_thread结构的内存。

既然要获得workerlock互斥量，当kill_workers函数正在遍历列表时，如果一个线程到达一个删除点，必须等待直到kill_workers释放互斥量时才可以继续处理。

792

```
550   /*
551    * Deal with signals.
552    *
553    * LOCKING: acquires and releases configlock.
554    */
555   void *
556   signal_thread(void *arg)
557   {
558       int     err, signo;

559       for (;;) {
560           err = sigwait(&mask, &signo);
561           if (err != 0)
562               log_quit("sigwait failed: %s", strerror(err));
563           switch (signo) {
564           case SIGHUP:
565               /*
566                * Schedule to re-read the configuration file.
567                */
568               pthread_mutex_lock(&configlock);
569               reread = 1;
570               pthread_mutex_unlock(&configlock);
571               break;

572           case SIGTERM:
573               kill_workers();
574               log_msg("terminate with signal %s", strsignal(signo));
575               exit(0);

576           default:
577               kill_workers();
578               log_quit("unexpected signal %d", signo);
579           }
580       }
581   }
```

[550-563] 函数signal_thread由负责处理信号的线程运行。在main函数中，初始化信号掩码，该掩码包括SIGHUP和SIGTERM。这里，调用sigwait来等待这些信号中的一个出现。如果sigwait失败，记录出错日志并退出。

[564-571] 如果接收到SIGHUP，就需要获得configlock互斥量，将reread变量设为1，然后释放互斥量。这就告诉打印守护进程程序在其处理循环的下一次迭代时再次读取配置文件。

[572-575] 如果接收到SIGTERM，调用kill_workers来杀掉所有的工作者线程，记录日志，然后调用exit终止进程。

[576-581] 如果接收到非期望的信号，则杀死工作者线程并调用log_quit来记录日志然后退出。

793

```
582   /*
583    * Add an option to the IPP header.
584    *
585    * LOCKING: none.
586    */
587   char *
588   add_option(char *cp, int tag, char *optname, char *optval)
589   {
```

```
590     int      n;
591     union {
592         int16_t s;
593         char c[2];
594     }        u;

595     *cp++ = tag;
596     n = strlen(optname);
597     u.s = htons(n);
598     *cp++ = u.c[0];
599     *cp++ = u.c[1];
600     strcpy(cp, optname);
601     cp += n;
602     n = strlen(optval);
603     u.s = htons(n);
604     *cp++ = u.c[0];
605     *cp++ = u.c[1];
606     strcpy(cp, optval);
607     return(cp + n);
608 }
```

[582-594] 函数add_option用于在送往打印机的IPP首部中添加选项。回忆图21-3，属性的格式是：描述属性的类型的1字节标志，然后是以2字节的二进制整数形式存储的属性名字的长度，接着是名字，属性值的大小，最后是属性值本身。

IPP没有意图去控制嵌入在首部的二进制整数的对齐方式。一些处理器架构（例如SPARC）并不能从任意地址装入一个整数。这意味着不能通过如下方式在IPP首部存放一个整数：该方式将一个指针转换成int16_t指向在首部存放整数的地址。代替地，需要一次复制1字节整数。这就是为什么定义一个包含16位整数和2字节数组的union。

[595-608] 在首部存储标志并将属性名字的长度转换为网络字节序。一次复制1个字节到首部。接着复制属性名字。重复这个过程，继续复制属性值，然后返回首部中下一部分应该开始的地址。

794

```
609 /*
610  * Single thread to communicate with the printer.
611  *
612  * LOCKING: acquires and releases joblock and configlock.
613  */
614 void *
615 printer_thread(void *arg)
616 {
617     struct job      *jp;
618     int             hlen, ilen, sockfd, fd, nr, nw;
619     char            *icp, *hcp;
620     struct ipp_hdr  *hp;
621     struct stat     sbuf;
622     struct iovec    iov[2];
623     char            name[FILENMSZ];
624     char            hbuf[HBUFSZ];
625     char            ibuf[IBUFSZ];
626     char            buf[IOBUFSZ];
627     char            str[64];

628     for (;;) {
629         /*
```

```
630          * Get a job to print.
631          */
632         pthread_mutex_lock(&joblock);
633         while (jobhead == NULL) {
634             log_msg("printer_thread: waiting...");
635             pthread_cond_wait(&jobwait, &joblock);
636         }
637         remove_job(jp = jobhead);
638         log_msg("printer_thread: picked up job %ld", jp->jobid);
639         pthread_mutex_unlock(&joblock);

640         update_jobno();
```

[609-627] 函数printer_thread由与网络打印机通信的线程运行。使用icp和ibuf来建立IPP首部。使用hcp和hbuf建立HTTP首部。需要在独立的缓冲区中建立首部。HTTP首部包括ASCII表示的长度字段，而且在拼装出IPP首部之前，并不知道应该预留多大的空间。在一次调用中使用writev来写这两个首部。

[628-640] 打印机线程运行在一个无限循环中，等待将作业传送到打印机。使用joblock互斥量来保护作业列表。如果作业没有挂起，使用pthread_cond_wait来等待到来的作业。当一个作业准备好时，调用remove_job将其从列表中删除。此时仍持有互斥量，因此释放互斥量并调用update_jobno将下一个作业号写入到/var/spool/printer/jobno。

795

```
641         /*
642          * Check for a change in the config file.
643          */
644         pthread_mutex_lock(&configlock);
645         if (reread) {
646             freeaddrinfo(printer);
647             printer = NULL;
648             printer_name = NULL;
649             reread = 0;
650             pthread_mutex_unlock(&configlock);
651             init_printer();
652         } else {
653             pthread_mutex_unlock(&configlock);
654         }

655         /*
656          * Send job to printer.
657          */
658         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jp->jobid);
659         if ((fd = open(name, O_RDONLY)) < 0) {
660             log_msg("job %ld canceled - can't open %s: %s",
661               jp->jobid, name, strerror(errno));
662             free(jp);
663             continue;
664         }
665         if (fstat(fd, &sbuf) < 0) {
666             log_msg("job %ld canceled - can't fstat %s: %s",
667               jp->jobid, name, strerror(errno));
668             free(jp);
669             close(fd);
670             continue;
671         }
```

[641-654] 由于有了要打印的作业，检查一下配置文件有无改变。锁住configlock并检查reread变量。如果该值非零，那么释放旧的addrinfo列表，指针设为空，解锁互斥量，然后调用init_printer来重新初始化指针信息。既然从main线程初始化后只有这个上下文中我们查看并可能更改打印机信息，我们除了使用configlock互斥量来保护reread标志的状态外，不需要任何其他的同步手段。注意，尽管在此函数中获得和释放两个不同互斥量，但是并没有同时持有两个互斥量，因此不需要建立一个锁层次（11.6节）。

[655-671] 如果不能打开数据文件，则记录日志，释放job结构，然后继续。打开文件之后，调用fstat来找到文件的大小，如果失败，记录日志并清理，然后继续。

796

```
672         if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
673             log_msg("job %ld deferred - can't create socket: %s",
674               jp->jobid, strerror(errno));
675             goto defer;
676         }
677         if (connect_retry(sockfd, printer->ai_addr,
678           printer->ai_addrlen) < 0) {
679             log_msg("job %ld deferred - can't contact printer: %s",
680               jp->jobid, strerror(errno));
681             goto defer;
682         }

683         /*
684          * Set up the IPP header.
685          */
686         icp = ibuf;
687         hp = (struct ipp_hdr *)icp;
688         hp->major_version = 1;
689         hp->minor_version = 1;
690         hp->operation = htons(OP_PRINT_JOB);
691         hp->request_id = htonl(jp->jobid);
692         icp += offsetof(struct ipp_hdr, attr_group);
693         *icp++ = TAG_OPERATION_ATTR;
694         icp = add_option(icp, TAG_CHARSET, "attributes-charset",
695           "utf-8");
696         icp = add_option(icp, TAG_NATULANG,
697           "attributes-natural-language", "en-us");
698         sprintf(str, "http://%s:%d", printer_name, IPP_PORT);
699         icp = add_option(icp, TAG_URI, "printer-uri", str);
```

[672-682] 打开一个流套接字与打印机通信。如果socket调用失败，跳到defer处，在那里将会清理、延迟一段时间然后再尝试。如果能够建立一个套接字，调用connect_retry连接打印机。

[683-699] 接下来，建立IPP首部。其操作是打印作业（print-job）请求。使用htons将2字节的操作ID从主机转为网络字节序，使用htonl将4字节的作业ID从主机转为网络字节序。完成首部的初始化之后，将设置标志值来显示其后跟随的操作属性。调用add_option来将属性添加到报文中。表21-1列出了打印作业请求必需的和可选的操作属性。前三个是必需的。将字符集设为UTF-8，该字符集是打印机必须支持的；指定语言为en-us，即代表美国英语（U.S. English）；另外一个必需的属性是URI（Universal Resource Identifier），将其设为

http://*printer_name*:631。(实际上应该向打印机要求获得其所支持的URI列表并从中选择一个，但那样做并没有增加太多价值，只是使本例变得复杂)。 797

```
700         icp = add_option(icp, TAG_NAMEWOLANG,
701           "requesting-user-name", jp->req.usernm);
702         icp = add_option(icp, TAG_NAMEWOLANG, "job-name",
703           jp->req.jobnm);
704         if (jp->req.flags & PR_TEXT) {
705             icp = add_option(icp, TAG_MIMETYPE, "document-format",
706               "text/plain");
707         } else {
708             icp = add_option(icp, TAG_MIMETYPE, "document-format",
709               "application/postscript");
710         }
711         *icp++ = TAG_END_OF_ATTR;
712         ilen = icp - ibuf;
713         /*
714          * Set up the HTTP header.
715          */
716         hcp = hbuf;
717         sprintf(hcp, "POST /%s/ipp HTTP/1.1\r\n", printer_name);
718         hcp += strlen(hcp);
719         sprintf(hcp, "Content-Length: %ld\r\n",
720           (long)sbuf.st_size + ilen);
721         hcp += strlen(hcp);
722         strcpy(hcp, "Content-Type: application/ipp\r\n");
723         hcp += strlen(hcp);
724         sprintf(hcp, "Host: %s:%d\r\n", printer_name, IPP_PORT);
725         hcp += strlen(hcp);
726         *hcp++ = '\r';
727         *hcp++ = '\n';
728         hlen = hcp - hbuf;
```

[700-712] 推荐使用requesting-user-name属性，但不是必需的。job-name属性也是可选的。print命令发送要打印的文件名作为作业名，该名字能够帮助用户区别多个要处理的作业。所提供的最后一个属性是document-format。如果省略该属性，则假定文件格式是打印机默认格式。对于PostScript打印机，格式可能是PostScript，但是一些打印机可以自动检测格式并在PostScript与文本或PostScript与PCL（HP的Printer Command Language）格式间做选择。如果设置了PR_TEXT标志，则将文档格式指定为text/plain；否则，设置为application/postscript。然后在属性结束处用属性结束标志定界并计算IPP首部的大小。

[713-728] 现在既然知道了IPP首部的大小，就可以建立HTTP首部。将Context-Length设为IPP首部的字节大小加上要打印文件的字节大小。Content-Type为application/ipp。用回车换行符结束HTTP首部。 798

```
729         /*
730          * Write the headers first.  Then send the file.
731          */
732         iov[0].iov_base = hbuf;
733         iov[0].iov_len = hlen;
734         iov[1].iov_base = ibuf;
```

```
735         iov[1].iov_len = ilen;
736         if ((nw = writev(sockfd, iov, 2)) != hlen + ilen) {
737             log_ret("can't write to printer");
738             goto defer;
739         }
740         while ((nr = read(fd, buf, IOBUFSZ)) > 0) {
741             if ((nw = write(sockfd, buf, nr)) != nr) {
742                 if (nw < 0)
743                   log_ret("can't write to printer");
744                 else
745                   log_msg("short write (%d/%d) to printer", nw, nr);
746                 goto defer;
747             }
748         }
749         if (nr < 0) {
750             log_ret("can't read %s", name);
751             goto defer;
752         }
753         /*
754          * Read the response from the printer.
755          */
756         if (printer_status(sockfd, jp)) {
757             unlink(name);
758             sprintf(name, "%s/%s/%ld", SPOOLDIR, REQDIR, jp->jobid);
759             unlink(name);
760             free(jp);
761             jp = NULL;
762         }
```

[729-739] 将iovec数组的第一个元素设为指向HTTP首部，第二个指向IPP首部。然后使用writev将两个首部送往打印机。如果写失败，记录日志并跳转到defer，在那里清理并延迟一段时间然后再尝试。

[740-752] 接下来，将数据文件发往打印机。把数据文件读入IOBUFSZ缓冲区块并写入与打印机相连的套接字。如果read或write失败，记录日志然后跳转到defer。

[753-762] 当整个文件发往打印机后，调用printer_status来接收打印机发回的对于打印请求的响应。如果printer_status成功，返回一个正值，删除数据和控制文件。然后释放job结构，将其指针设为NULL，并到达defer标签。

799

```
763     defer:
764             close(fd);
765             if (sockfd >= 0)
766                 close(sockfd);
767             if (jp != NULL) {
768                 replace_job(jp);
769                 sleep(60);
770             }
771         }
772     }
773     /*
774      * Read data from the printer, possibly increasing the buffer.
775      * Returns offset of end of data in buffer or -1 on failure.
776      *
777      * LOCKING: none.
778      */
779     ssize_t
```

```
780  readmore(int sockfd, char **bpp, int off, int *bszp)
781  {
782      ssize_t nr;
783      char    *bp = *bpp;
784      int     bsz = *bszp;

785      if (off >= bsz) {
786          bsz += IOBUFSZ;
787          if ((bp = realloc(*bpp, bsz)) == NULL)
788              log_sys("readmore: can't allocate bigger read buffer");
789          *bszp = bsz;
790          *bpp = bp;
791      }
792      if ((nr = tread(sockfd, &bp[off], bsz-off, 1)) > 0)
793          return(off+nr);
794      else
795          return(-1);
796  }
```

[763-772] 在defer标签处，关闭打开的数据文件的文件描述符。如果套接字描述符是有效的，则关闭之。如果出错，将作业放回挂起的作业列表的头部，然后延迟1分钟。如果成功，jp为NULL，因此简单地回到循环开始处，以获得下一个要打印的作业。

[773-796] readmore函数用于读取来自打印机的部分响应消息。如果到达缓冲区尾部，通过相应的参数bpp和bszp重新分配一个大一点的缓冲区并返回该新的缓冲区的起始地址以及缓冲区大小。上述任何一种情况下，从缓冲区已读数据的末尾开始读取缓冲区所能容纳的尽可能多的数据。返回相应的已读数据末尾的新偏移量。如果read失败，或者超时，返回-1。

800

```
797  /*
798   * Read and parse the response from the printer.  Return 1
799   * if the request was successful, and 0 otherwise.
800   *
801   * LOCKING: none.
802   */
803  int
804  printer_status(int sockfd, struct job *jp)
805  {
806      int             i, success, code, len, found, bufsz;
807      long            jobid;
808      ssize_t         nr;
809      char            *statcode, *reason, *cp, *contentlen;
810      struct ipp_hdr  *hp;
811      char            *bp;

812      /*
813       * Read the HTTP header followed by the IPP response header.
814       * They can be returned in multiple read attempts.  Use the
815       * Content-Length specifier to determine how much to read.
816       */
817      success = 0;
818      bufsz = IOBUFSZ;
819      if ((bp = malloc(IOBUFSZ)) == NULL)
820          log_sys("printer_status: can't allocate read buffer");

821      while ((nr = tread(sockfd, bp, IOBUFSZ, 5)) > 0) {
822          /*
```

```
823          * Find the status.  Response starts with "HTTP/x.y"
824          * so we can skip the first 8 characters.
825          */
826         cp = bp + 8;
827         while (isspace((int)*cp))
828             cp++;
829         statcode = cp;
830         while (isdigit((int)*cp))
831             cp++;
832         if (cp == statcode) { /* Bad format; log it and move on */
833             log_msg(bp);
```

[797-811] printer_status函数读取打印机对一个打印作业请求的响应消息。不知道打印机会如何响应：也许会在多个报文里回送一个响应；也许在一个报文里回送完整的响应；或者包括一个中间确认，如HTTP 100 Continue报文。需要处理所有的可能性。

[812-833] 分配一个缓冲区并读取来自打印机的数据，期望5秒之内有可用的响应。跳过HTTP/1.1和报文开始的所有空格，其后应该是数字的状态码。如果不是，在日志中记录报文的内容。

801

```
834         } else {
835             *cp++ = '\0';
836             reason = cp;
837             while (*cp != '\r' && *cp != '\n')
838                 cp++;
839             *cp = '\0';
840             code = atoi(statcode);
841             if (HTTP_INFO(code))
842                 continue;
843             if (!HTTP_SUCCESS(code)) { /* probable error: log it */
844                 bp[nr] = '\0';
845                 log_msg("error: %s", reason);
846                 break;
847             }

848             /*
849              * The HTTP request was okay, but we still
850              * need to check the IPP status.  First
851              * search for the Content-Length specifier.
852              */
853             i = cp - bp;
854             for (;;) {
855                 while (*cp != 'C' && *cp != 'c' && i < nr) {
856                     cp++;
857                     i++;
858                 }
859                 if (i >= nr &&  /* get more header */
860                   ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
861                     goto out;
862                 cp = &bp[i];
```

[834-839] 如果在响应中找到一个数字状态码，将其开始的非数字字符转换成空字节'\0'。其后应该跟随一个表明出错原因的字符串（文本消息）。搜索回车或换行符，并采用空字节'\0'结束文本字符串。

[840-847] 将代码转换为整数。如果仅是提供信息的报文，将其忽略并继续循环。期望看

到要么是一个成功消息要么是一个出错消息。如果得到一个出错消息，记录出错日志并退出循环。

[848-862] 如果HTTP请求成功，需要检查IPP状态。搜索整个报文直到找到Content-Length属性，然后查看C或c。HTTP首部的关键字是大小写敏感的，因此需要同时检查小写字符和大写字符。

如果缓冲区空间耗尽，需要再次读。既然readmore调用realloc，可能会改变缓冲区的地址，需要重置cp以指向缓冲区合适的地方。

802

```
863                 if (strncasecmp(cp, "Content-Length:", 15) == 0) {
864                     cp += 15;
865                     while (isspace((int)*cp))
866                         cp++;
867                     contentlen = cp;
868                     while (isdigit((int)*cp))
869                         cp++;
870                     *cp++ = '\0';
871                     i = cp - bp;
872                     len = atoi(contentlen);
873                     break;
874                 } else {
875                     cp++;
876                     i++;
877                 }
878             }
879             if (i >= nr &&  /* get more header */
880               ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
881                 goto out;
882             cp = &bp[i];

883             found = 0;
884             while (!found) {  /* look for end of HTTP header */
885                 while (i < nr - 2) {
886                     if (*cp == '\n' && *(cp + 1) == '\r' &&
887                       *(cp + 2) == '\n') {
888                         found = 1;
889                         cp += 3;
890                         i += 3;
891                         break;
892                     }
893                     cp++;
894                     i++;
895                 }
896                 if (i >= nr &&  /* get more header */
897                   ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
898                     goto out;
899                 cp = &bp[i];
900             }
```

[863-882] 如果找到Content-Length属性字符串，搜索其值。将这个数字字符串转换成整数，并退出for循环，如果耗尽缓冲区那么从打印机再次读取。如果到达缓冲区尾也没有找到Content-Length属性，继续循环并从打印机处再次读取。

[883-900] 一旦获得Content-Length所指定的报文长度，搜索HTTP首部的结束部分(一个空白行)。如果找到，则设置found标志并跳过报文中的空白行。

803

```
901             if (nr - i < len && /* get more header */
902               ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
903                 goto out;
904             cp = &bp[i];

905             hp = (struct ipp_hdr *)cp;
906             i = ntohs(hp->status);
907             jobid = ntohl(hp->request_id);
908             if (jobid != jp->jobid) {
909                 /*
910                  * Different jobs.  Ignore it.
911                  */
912                 log_msg("jobid %ld status code %d", jobid, i);
913                 break;
914             }

915             if (STATCLASS_OK(i))
916                 success = 1;
917             break;
918         }
919     }

920 out:
921     free(bp);
922     if (nr < 0) {
923         log_msg("jobid %ld: error reading printer response: %s",
924           jobid, strerror(errno));
925     }
926     return(success);
927 }
```

[901-904] 继续搜索HTTP首部的结尾。如果耗尽缓冲区空间，再次读取。如果找到HTTP首部的结尾，计算HTTP首部所用的字节数。如果所读取的数据大小减去HTTP首部的大小后不等于IPP报文的数据长度（该值从内容长度Content-Length中计算），就需要再次读取。

[905-927] 从报文IPP首部中获取状态和作业ID。两者均以网络字节序的整数形式存储，因此需要调用ntohs和ntohl将其转换为主机字节序。如果作业ID不匹配，表明并非这里的响应，因此记录日志并退出外部while循环。如果IPP指示为成功状态，保存返回值并退出循环。如果打印请求成功返回1，如果失败则返回0。

这里总结本章中这个扩展的例子。本章中的程序在Xerox Phaser 860网络PostScript打印机上测试。遗憾的是，这种打印机不能识别text/plain文档格式，但是确实能够自动检查文本和PostScript。因此，这种打印机上可以打印PostScript文件和文本文件，但是不能按照文本那样打印PostScript源文件，除非使用另外的工具（如a2ps(1)）来封装PostScript程序。

804

21.6 小结

本章仔细考查了两个完整的程序：一个打印假脱机守护进程，可以将打印作业发送到网络打印机；一个命令行程序，将要打印的作业提交到假脱机守护进程。这提供了一个机会，考查在一个实际程序中使用前面章节所讲述的许多特性，例如：线程、I/O多路技术、文件I/O、套接字I/O及信号。

习题

21.1 将ipp.h中所列的IPP错误代码值转换成错误消息。然后修改打印假脱机守护进程，当IPP首部指示有打印机错误时，在printer_status函数结尾处记录日志。

21.2 增强print命令和printd守护进程，使得用户可以请求双面打印。同样，使其可以支持横向打印和纵向打印。

21.3 修改打印假脱机守护进程，使得当其开始时，能够联系打印机并找出打印机所支持的特性，这样守护进程就不会请求打印机所不支持的选项。

21.4 写一个命令行程序来报告挂起的打印作业状态。

21.5 写一个命令行程序来取消一个挂起的打印作业。

21.6 在打印假脱机守护进程中支持多个打印机，并包括将打印作业从一个打印机移到另一个打印机的方式。

805

附录A

函数原型

本附录包含了正文中描述过的标准ISO C、POSIX和UNIX系统函数的函数原型。通常我们想了解的只是函数的参数（fgets的哪一个参数是文件指针？）或者返回值（sprintf返回的是指针还是计数值？）。这些函数原型还说明了要包含哪些头文件，以获得特殊常量的定义，或获得ISO C函数原型，以帮助检测编译时错误。

引用每个函数原型的页号出现在为该函数列出的第一个头文件的右边，这个页号指的是包含该函数原型的页。为获得该函数原型的附加信息可参阅该页。

某些函数原型仅受本书描述的几种平台中某几种的支持。另外，某些函数标志是有些平台支持而另一些平台并不支持的。对于这些情况，我们通常列出提供支持的平台，但是对于少数情况，我们列出了不提供支持的平台。

```
void        abort(void);
                        <stdlib.h>                                    p.274
                        此函数不返回

int         accept(int sockfd, struct sockaddr *restrict addr,
                   socklen_t *restrict len);
                        <sys/socket.h>                                p.451
                        返回值：若成功则返回文件（套接字）描述符，若出错则返回-1

int         access(const char *pathname, int mode);
                        <unistd.h>                                    p.78
                        mode: R_OK, W_OK, X_OK, F_OK
                        返回值：若成功则返回0，若出错则返回-1
unsigned
int         alarm(unsigned int seconds);
                        <unistd.h>                                    p.252
                        返回值：0或以前设置的闹钟时间的余留秒数

char       *asctime(const struct tm *tmptr);
                        <time.h>                                      p.144
                        返回值：指向以null结尾的字符串的指针

int         atexit(void (*func)(void));
                        <stdlib.h>                                    p.149
                        返回值：若成功则返回0，若出错则返回非0值

int         bind(int sockfd, const struct sockaddr *addr, socklen_t len);
                        <sys/socket.h>                                p.449
                        返回值：若成功则返回0，若出错则返回-1

void       *calloc(size_t nobj, size_t size);
                        <stdlib.h>                                    p.155
                        返回值：若成功则返回非空指针，若出错则返回NULL
```

speed_t **cfgetispeed**(const struct termios **termptr*);
<termios.h> p.523
返回值：波特率值

speed_t **cfgetospeed**(const struct termios **termptr*);
<termios.h> p.523
返回值：波特率值

int **cfsetispeed**(struct termios **termptr*, speed_t *speed*);
<termios.h> p.523
返回值：若成功则返回0，若出错则返回-1

int **cfsetospeed**(struct termios **termptr*, speed_t *speed*);
<termios.h> p.523
返回值：若成功则返回0，若出错则返回-1

int **chdir**(const char **pathname*);
<unistd.h> p.102
返回值：若成功则返回0，若出错则返回-1

int **chmod**(const char **pathname*, mode_t *mode*);
<sys/stat.h> p.81
mode：S_IS[UG]ID，S_ISVTX，S_I[RWX](USR|GRP|OTH)
返回值：若成功则返回0，若出错则返回-1

808

int **chown**(const char **pathname*, uid_t *owner*, gid_t *group*);
<unistd.h> p.84
返回值：若成功则返回0，若出错则返回-1

void **clearerr**(FILE **fp*);
<stdio.h> p.115

int **close**(int *filedes*);
<unistd.h> p.50
返回值：若成功则返回0，若出错则返回-1

int **closedir**(DIR **dp*);
<dirent.h> p.98
返回值：若成功则返回0，若出错则返回-1

void **closelog**(void);
<syslog.h> p.346

unsigned char ***CMSG_DATA**(struct cmsghdr **cp*);
<sys/socket.h> p.487
返回值：指向与cmsghdr结构相关联的数据的指针

struct cmsghdr ***CMSG_FIRSTHDR**(struct msghdr **mp*);
<sys/socket.h> p.487
返回值：指向与msghdr结构相关联的第一个cmsghdr结构的指针，若无这样的结构则返回NULL

unsigned int **CMSG_LEN**(unsigned int *nbytes*);
<sys/socket.h> p.487
返回值：为*nbytes*大小的数据对象分配的长度

struct cmsghdr ***CMSG_NXTHDR**(struct msghdr **mp*, struct cmsghdr **cp*);
<sys/socket.h> p.487
返回值：指向与msghdr结构相关联的下一个cmsghdr结构的指针，该msghdr结构给出了当前cmsghdr结构，若当前cmsghdr结构已是最后一个则返回NULL

int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
<sys/socket.h> p.450
返回值：若成功则返回0，若出错则返回−1

int creat(const char *pathname, mode_t mode);
<fcntl.h> p.50
mode：S_IS[UG]ID，S_ISVTX，S_I[RWX](USR|GRP|OTH)
返回值：若成功则返回为只写打开的文件描述符，若出错则返回−1

char *ctermid(char *ptr);
<stdio.h> p.524
返回值：若成功则返回指向控制终端名的指针，若出错则返回指向空字符串的指针

809

char *ctime(const time_t *calptr);
<time.h> p.144
返回值：指向以null结尾的字符串的指针

int dup(int filedes);
<unistd.h> p.60
返回值：若成功则返回新的文件描述符，若出错则返回−1

int dup2(int filedes, int filedes2);
<unistd.h> p.60
返回值：若成功则返回新的文件描述符，若出错则返回−1

void endgrent(void);
<grp.h> p.137

void endhostent(void);
<netdb.h> p.443

void endnetent(void);
<netdb.h> p.443

void endprotoent(void);
<netdb.h> p.444

void endpwent(void);
<pwd.h> p.135

void endservent(void);
<netdb.h> p.444

void endspent(void);
<shadow.h> p.137
平台：Linux 2.4.22，Solaris 9

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
<unistd.h> p.188
返回值：若出错则返回−1，若成功则不返回值

int execle(const char *pathname, const char *arg0, ... /* (char *) 0,
char *const envp[] */);
<unistd.h> p.188
返回值：若出错则返回−1，若成功则不返回值

int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
<unistd.h> p.188
返回值：若出错则返回−1，若成功则不返回值

810

int **execv**(const char *pathname, char *const argv[]);
<unistd.h>
返回值：若出错则返回-1，若成功则不返回值 p.188

int **execve**(const char *pathname, char *const argv[], char *const envp[]);
<unistd.h>
返回值：若出错则返回-1，若成功则不返回值 p.188

int **execvp**(const char *filename, char *const argv[]);
<unistd.h>
返回值：若出错则返回-1，若成功则不返回值 p.188

void **_Exit**(int status);
<stdlib.h>
此函数不返回 p.148

void **_exit**(int status);
<unistd.h>
此函数不返回 p.148

void **exit**(int status);
<stdlib.h>
此函数不返回 p.148

int **fattach**(int filedes, const char *path);
<stropts.h>
返回值：若成功则返回0，若出错则返回-1 p.472
平台：Linux 2.4.22，Solaris 9

int **fchdir**(int filedes);
<unistd.h>
返回值：若成功则返回0，若出错则返回-1 p.102

int **fchmod**(int filedes, mode_t mode);
<sys/stat.h>
mode：S_IS[UG]ID，S_ISVTX，S_I[RWX](USR|GRP|OTH) p.81
返回值：若成功则返回0，若出错则返回-1

int **fchown**(int filedes, uid_t owner, gid_t group);
<unistd.h>
返回值：若成功则返回0，若出错则返回-1 p.84

int **fclose**(FILE *fp);
<stdio.h>
返回值：若成功则返回0，若出错则返回EOF p.114

int **fcntl**(int filedes, int cmd, ... /* int arg */);
<fcntl.h>
cmd：F_DUPFD，F_GETFD，F_SETFD，F_GETFL，F_SETFL，
F_GETOWN，F_SETOWN，F_GETLK，F_SETLK，F_SETLKW p.62
返回值：若成功则依赖于cmd，若出错则返回-1

811

int **fdatasync**(int filedes);
<unistd.h>
返回值：若成功则返回0，若出错则返回-1 p.62
平台：Linux 2.4.22，Solaris 9

```
void    FD_CLR(int fd, fd_set *fdset);
                <sys/select.h>                                          p.382

int     fdetach(const char *path);
                <stropts.h>                                             p.472
                返回值：若成功则返回0，若出错则返回-1
                平台：Linux 2.4.22，Solaris 9

int     FD_ISSET(int fd, fd_set *fdset);
                <sys/select.h>                                          p.382
                返回值：若fd在描述符集中则返回非0值，否则返回0

FILE    *fdopen(int filedes, const char *type);
                <stdio.h>                                               p.112
                type: "r", "w", "a", "r+", "w+", "a+",
                返回值：若成功则返回文件指针，若出错则返回NULL

void    FD_SET(int fd, fd_set *fdset);
                <sys/select.h>                                          p.382

void    FD_ZERO(fd_set *fdset);
                <sys/select.h>                                          p.382

int     feof(FILE *fp);
                <stdio.h>                                               p.115
                返回值：若在流上文件结尾则返回非0值（真），否则返回0（假）

int     ferror(FILE *fp);
                <stdio.h>                                               p.115
                返回值：若在流上出错则返回非0值（真），否则返回0（假）

int     fflush(FILE *fp);
                <stdio.h>                                               p.112
                返回值：若成功则返回0，若出错则返回EOF

int     fgetc(FILE *fp);
                <stdio.h>                                               p.115
                返回值：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF

int     fgetpos(FILE *restrict fp, fpos_t *restrict pos);
                <stdio.h>                                               p.121
                返回值：若成功则返回0，若出错则返回非0值

char    *fgets(char *restrict buf, int n, FILE *restrict fp);
                <stdio.h>                                               p.116
                返回值：若成功则返回buf，若已到达文件结尾或出错则返回NULL

int     fileno(FILE *fp);
                <stdio.h>                                               p.125
                返回值：与该流相关联的文件描述符

void    flockfile(FILE *fp);
                <stdio.h>                                               p.325

FILE    *fopen(const char *restrict pathname, const char *restrict type);
                <stdio.h>                                               p.112
                type: "r", "w", "a", "r+", "w+", "a+",
                返回值：若成功则返回文件指针，若出错则返回NULL
```

812

```
pid_t    fork(void);
                        <unistd.h>                                              p.172
                        返回值：子进程中返回0，父进程中返回子进程ID，出错返回-1

long     fpathconf(int filedes, int name);
                        <unistd.h>                                              p.33
                        name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,
                              _PC_FILESIZEBITS, _PC_LINK_MAX, _PC_MAX_CANON,
                              _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
                              _PC_PATH_MAX, 'u'_PC_PIPE_BUF, _PC_PRIO_IO,
                              _PC_SYNC_IO, _PC_SYMLINK_MAX, _PC_VDISABLE
                        返回值：若成功则返回相应值，若出错则返回-1

int      fprintf(FILE *restrict fp, const char *restrict format, ...);
                        <stdio.h>                                               p.121
                        返回值：若成功则返回输出字符数，若输出出错则返回负值

int      fputc(int c, FILE *fp);
                        <stdio.h>                                               p.116
                        返回值：若成功则返回c，若出错则返回EOF

int      fputs(const char *restrict str, FILE *restrict fp);
                        <stdio.h>                                               p.117
                        返回值：若成功则返回非负值，若出错则返回EOF

size_t   fread(void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);
                        <stdio.h>                                               p.119
                        返回值：读的对象数

void     free(void *ptr);
                        <stdlib.h>                                              p.155

void     freeaddrinfo(struct addrinfo *ai);
                        <sys/socket.h>                                          p.445
                        <netdb.h>

FILE     *freopen(const char *restrict pathname, const char *restrict type,
                  FILE *restrict fp);
                        <stdio.h>                                               p.112
                        type: "r", "w", "a", "r+", "w+", "a+",
                        返回值：若成功则返回文件指针，若出错则返回NULL

int      fscanf(FILE *restrict fp, const char *restrict format, ...);
                        <stdio.h>                                               p.124
                        返回值：指定的输入项数，若输入出错或在任意变换前已到达文件结尾则返回EOF

int      fseek(FILE *fp, long offset, int whence);
                        <stdio.h>                                               p.120
                        whence: SEEK_SET, SEEK_CUR, SEEK_END
                        返回值：若成功则返回0，若出错则返回非0值

int      fseeko(FILE *fp, off_t offset, int whence);
                        <stdio.h>                                               p.121
                        whence: SEEK_SET, SEEK_CUR, SEEK_END
                        返回值：若成功则返回0，若出错则返回非0值
```

813

int **fsetpos**(FILE *fp, const fpos_t *pos);
<stdio.h> p.121
返回值：若成功则返回0，若出错则返回非0值

int **fstat**(int filedes, struct stat *buf);
<sys/stat.h> p.71
返回值：若成功则返回0，若出错则返回−1

int **fsync**(int filedes);
<unistd.h> p.62
返回值：若成功则返回0，若出错则返回−1

long **ftell**(FILE *fp);
<stdio.h> p.120
返回值：若成功则返回当前文件位置指示，若出错则返回−1L

off_t **ftello**(FILE *fp);
<stdio.h> p.121
返回值：若成功则返回当前文件位置指示，若出错则返回(off_t)-1

key_t **ftok**(const char *path, int id);
<sys/ipc.h> p.416
返回值：若成功则返回键，若出错则返回(key_t)-1

814

int **ftruncate**(int filedes, off_t length);
<unistd.h> p.86
返回值：若成功则返回0，若出错则返回−1

int **ftrylockfile**(FILE *fp);
<stdio.h> p.325
返回值：若成功则返回0，否则返回非0值

void **funlockfile**(FILE *fp);
<stdio.h> p.325

int **fwide**(FILE *fp, int mode);
<stdio.h> p.109
<wchar.h>
返回值：若流是宽定向的则返回正值，若流是字节定向的则返回负值，或者若流是未定向的则返回0

size_t **fwrite**(const void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);
<stdio.h> p.119
返回值：写的对象数

const char ***gai_strerror**(int error);
<netdb.h> p.446
返回值：指向描述错误的字符串的指针

int **getaddrinfo**(const char *restrict host, const char *restrict service, const struct addrinfo *restrict hint, struct addrinfo **restrict res);
<sys/socket.h> p.445
<netdb.h>
返回值：若成功则返回0，若出错则返回非0错误码

int **getc**(FILE **fp*);
<stdio.h> p.115
返回值：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF

int **getchar**(void);
<stdio.h> p.115
返回值：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF

int **getchar_unlocked**(void);
<stdio.h> p.326
返回值：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF

int **getc_unlocked**(FILE **fp*);
<stdio.h> p.326
返回值：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF

815

char ***getcwd**(char **buf*, size_t *size*);
<unistd.h> p.103
返回值：若成功则返回*buf*，若出错则返回NULL

gid_t **getegid**(void);
<unistd.h> p.172
返回值：调用进程的有效组ID

char ***getenv**(const char **name*);
<stdlib.h> p.157
返回值：指向与*name*关联的*value*的指针，若未找到则返回NULL

uid_t **geteuid**(void);
<unistd.h> p.172
返回值：调用进程的有效用户ID

gid_t **getgid**(void);
<unistd.h> p.172
返回值：调用进程的实际组ID

struct group ***getgrent**(void);
<grp.h> p.137
返回值：若成功则返回指针，若出错或到达文件结尾则返回NULL

struct group ***getgrgid**(gid_t *gid*);
<grp.h> p.137
返回值：若成功则返回指针，若出错则返回NULL

struct group ***getgrnam**(const char **name*);
<grp.h> p.137
返回值：若成功则返回指针，若出错则返回NULL

int **getgroups**(int *gidsetsize*, gid_t *grouplist*[]);
<unistd.h> p.138
返回值：若成功则返回附加组ID数，若出错则返回-1

struct hostent ***gethostent**(void);
<netdb.h> p.443
返回值：若成功则返回指针，若出错则返回NULL

int **gethostname**(char **name*, int *namelen*);
<unistd.h> p.142
返回值：若成功则返回0，若出错则返回-1

```
char    *getlogin(void);
                    <unistd.h>                                          p.208
                    返回值：若成功则返回指向登录名字符串的指针，若出错则返回NULL
```

816

```
int     getmsg(int filedes, struct strbuf *restrict ctlptr,
               struct strbuf *restrict dataptr, int *restrict flagptr);
                    <stropts.h>                                         p.377
                    *flagptr: 0, RS_HIPRI
                    返回值：若成功则返回非负值，若出错则返回-1
                    平台：Linux 2.4.22, Solaris 9
```

```
int     getnameinfo(const struct sockaddr *restrict addr, socklen_t alen,
                    char *restrict host, socklen_t hostlen, char *restrict service,
                    socklen_t servlen, unsigned int flags);
                    <sys/socket.h>                                      p.446
                    <netdb.h>
                    返回值：若成功则返回0，若出错则返回非0值
```

```
struct
netent  *getnetbyaddr(uint32_t net, int type);
                    <netdb.h>                                           p.443
                    返回值：若成功则返回指针，若出错则返回NULL
```

```
struct
netent  *getnetbyname(const char *name);
                    <netdb.h>                                           p.443
                    返回值：若成功则返回指针，若出错则返回NULL
```

```
struct
netent  *getnetent(void);
                    <netdb.h>                                           p.443
                    返回值：若成功则返回指针，若出错则返回NULL
```

```
int     getopt(int argc, const * const argv[], const char *options);
                    <fcntl.h>                                           p.619
                    extern int optind, opterr, optopt;
                    extern char *optarg;
                    返回值：下一个选项字符，若全部选项处理完毕则返回-1
```

```
int     getpeername(int sockfd, struct sockaddr *restrict addr,
                    socklen_t *restrict alenp);
                    <sys/socket.h>                                      p.450
                    返回值：若成功则返回0，若出错则返回-1
```

```
pid_t   getpgid(pid_t pid);
                    <unistd.h>                                          p.218
                    返回值：若成功则返回进程组ID，若出错则返回-1
```

```
pid_t   getpgrp(void);
                    <unistd.h>                                          p.218
                    返回值：调用进程的进程组ID
```

```
pid_t   getpid(void);
                    <unistd.h>                                          p.171
                    返回值：调用进程的进程ID
```

817

```
int     getpmsg(int filedes, struct strbuf *restrict ctlptr,
                struct strbuf *restrict dataptr, int *restrict bandptr,
                int *restrict flagptr);
                    <stropts.h>                                         p.377
                    *flagptr: 0, MSG_HIPRI, MSG_BAND, MSG_ANY
                    返回值：若成功则返回非负值，若出错则返回-1
                    平台：Linux 2.4.22, Solaris 9
```

```
pid_t     getppid(void);
                          <unistd.h>                                          p.171
                          返回值：调用进程的父进程ID

struct
protoent *getprotobyname(const char *name);
                          <netdb.h>                                           p.444
                          返回值：若成功则返回指针，若出错则返回NULL

struct
protoent *getprotobynumber(int proto);
                          <netdb.h>                                           p.444
                          返回值：若成功则返回指针，若出错则返回NULL

struct
protoent *getprotoent(void);
                          <netdb.h>                                           p.444
                          返回值：若成功则返回指针，若出错则返回NULL

struct
passwd   *getpwent(void);
                          <pwd.h>                                             p.135
                          返回值：若成功则返回指针，若出错或到达文件结尾则返回NULL

struct
passwd   *getpwnam(const char *name);
                          <pwd.h>                                             p.135
                          返回值：若成功则返回指针，若出错则返回NULL

struct
passwd   *getpwuid(uid_t uid);
                          <pwd.h>                                             p.135
                          返回值：若成功则返回指针，若出错则返回NULL

int       getrlimit(int resource, struct rlimit *rlptr);
                          <sys/resource.h>                                    p.165
                          返回值：若成功则返回0，若出错则返回非0值

char     *gets(char *buf);
                          <stdio.h>                                           p.116
                          返回值：若成功则返回buf，若已到达文件结尾或出错则返回NULL

struct
servent  *getservbyname(const char *name, const char *proto);
                          <netdb.h>                                           p.444
                          返回值：若成功则返回指针，若出错则返回NULL

struct
servent  *getservbyport(int port, const char *proto);
                          <netdb.h>                                           p.444
                          返回值：若成功则返回指针，若出错则返回NULL

struct
servent  *getservent(void);
                          <netdb.h>                                           p.444
                          返回值：若成功则返回指针，若出错则返回NULL

pid_t     getsid(pid_t pid);
                          <unistd.h>                                          p.220
                          返回值：若成功则返回会话首进程的进程组ID，若出错则返回-1

int       getsockname(int sockfd, struct sockaddr *restrict addr,
                          socklen_t *restrict alenp);
                          <sys/socket.h>                                      p.449
                          返回值：若成功则返回0，若出错则返回-1
```

818

```
int        getsockopt(int sockfd, int level, int option, void *restrict val,
                      socklen_t *restrict lenp);
                        <sys/socket.h>                                        p.465
```
返回值：若成功则返回0，若出错则返回-1

```
struct
spwd      *getspent(void);
                        <shadow.h>                                            p.137
```
返回值：若成功则返回指针，若出错则返回NULL
平台：Linux 2.4.22，Solaris 9

```
struct
spwd      *getspnam(const char *name);
                        <shadow.h>                                            p.137
```
返回值：若成功则返回指针，若出错则返回NULL
平台：Linux 2.4.22，Solaris 9

```
int        gettimeofday(struct timeval *restrict tp, void *restrict tzp);
                        <sys/time.h>                                          p.142
```
返回值：总是返回0

```
uid_t      getuid(void);
                        <unistd.h>                                            p.172
```
返回值：调用进程的实际用户ID

```
struct
tm        *gmtime(const time_t *calptr);
                        <time.h>                                              p.144
```
返回值：指向tm结构的指针

```
int        grantpt(int filedes);
                        <stdlib.h>                                            p.545
```
返回值：若成功则返回0，若出错则返回-1
平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

819

```
uint32_t   htonl(uint32_t hostint32);
                        <arpa/inet.h>                                         p.440
```
返回值：以网络字节序表示的32位整型数

```
uint16_t   htons(uint16_t hostint16);
                        <arpa/inet.h>                                         p.440
```
返回值：以网络字节序表示的16位整型数

```
const
char      *inet_ntop(int domain, const void *restrict addr, char *restrict str,
                     socklen_t size);
                        <arpa/inet.h>                                         p.442
```
返回值：若成功则返回地址字符串指针，若出错则返回NULL

```
int        inet_pton(int domain, const char *restrict str, void *restrict addr);
                        <arpa/inet.h>                                         p.442
```
返回值：若成功则返回1，若格式无效则返回0，若出错则返回-1

```
int        initgroups(const char *username, gid_t basegid);
                        <grp.h>     /* Linux & Solaris */                     p.138
                        <unistd.h> /* FreeBSD & Mac OS X */
```
返回值：若成功则返回0，若出错则返回-1

```
int        ioctl(int filedes, int request, ...);
                        <unistd.h>      /* System V */                        p.67
                        <sys/ioctl.h>   /* BSD and Linux */
                        <stropts.h>     /* XSI STREAMS */
```
返回值：若出错则返回-1，若成功则返回其他值

int **isastream**(int *filedes*);
<stropts.h> p.373
返回值：若为STREAMS设备则返回1（真），否则返回0（假）
平台：Linux 2.4.22，Solaris 9

int **isatty**(int *filedes*);
<unistd.h> p.525
返回值：若为终端设备则返回1（真），否则返回0（假）

int **kill**(pid_t *pid*, int *signo*);
<signal.h> p.251
返回值：若成功则返回0，若出错则返回-1

int **lchown**(const char **pathname*, uid_t *owner*, gid_t *group*);
<unistd.h> p.84
返回值：若成功则返回0，若出错则返回-1

int **link**(const char **existingpath*, const char **newpath*);
<unistd.h> p.89
返回值：若成功则返回0，若出错则返回-1

820

int **listen**(int *sockfd*, int *backlog*);
<sys/socket.h> p.451
返回值：若成功则返回0，若出错则返回-1

struct tm ***localtime**(const time_t **calptr*);
<time.h> p.144
返回值：指向tm结构的指针

void **longjmp**(jmp_buf *env*, int *val*);
<setjmp.h> p.161
此函数不返回

off_t **lseek**(int *filedes*, off_t *offset*, int *whence*);
<unistd.h> p.50
whence: SEEK_SET, SEEK_CUR, SEEK_END
返回值：若成功则返回新的文件偏移量，若出错则返回-1

int **lstat**(const char *restrict *pathname*, struct stat *restrict *buf*);
<sys/stat.h> p.71
返回值：若成功则返回0，若出错则返回-1

void ***malloc**(size_t *size*);
<stdlib.h> p.155
返回值：若成功则返回非空指针，若出错则返回NULL

int **mkdir**(const char **pathname*, mode_t *mode*);
<sys/stat.h> p.97
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
返回值：若成功则返回0，若出错则返回-1

int **mkfifo**(const char **pathname*, mode_t *mode*);
<sys/stat.h> p.412
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
返回值：若成功则返回0，若出错则返回-1

int **mkstemp**(char **template*);
<stdlib.h> p.129
返回值：若成功则返回文件描述符，若出错则返回-1

```
time_t    mktime(struct tm *tmptr);
                    <time.h>                                                p.144
                    返回值：若成功则返回日历时间，若出错则返回-1

caddr_t   *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);
                    <sys/mman.h>                                            p.391
                    prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
                    flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
                    返回值：若成功则返回映射区的起始地址，若出错则返回MAP_FAILED
```

821

```
int       mprotect(void *addr, size_t len, int prot);
                    <sys/mman.h>                                            p.393
                    返回值：若成功则返回0，若出错则返回-1

int       msgctl(int msqid, int cmd, struct msqid_ds *buf);
                    <sys/msg.h>                                             p.420
                    cmd: IPC_STAT, IPC_SET, IPC_RMID
                    返回值：若成功则返回0，若出错则返回-1
                    平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

int       msgget(key_t key, int flag);
                    <sys/msg.h>                                             p.419
                    flag: 0, IPC_CREAT, IPC_EXCL
                    返回值：若成功则返回消息队列ID，若出错则返回-1
                    平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

ssize_t   msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                    <sys/msg.h>                                             p.421
                    flag: 0, IPC_NOWAIT, MSG_NOERROR
                    返回值：若成功则返回消息的数据部分的长度，若出错则返回-1
                    平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

int       msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
                    <sys/msg.h>                                             p.420
                    flag: 0, IPC_NOWAIT
                    返回值：若成功则返回0，若出错则返回-1
                    平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

int       msync(void *addr, size_t len, int flags);
                    <sys/mman.h>                                            p.393
                    返回值：若成功则返回0，若出错则返回-1

int       munmap(caddr_t addr, size_t len);
                    <sys/mman.h>                                            p.393
                    返回值：若成功则返回0，若出错则返回-1

uint32_t  ntohl(uint32_t netint32);
                    <arpa/inet.h>                                           p.440
                    返回值：以主机字节序表示的32位整型数

uint16_t  ntohs(uint16_t netint16);
                    <arpa/inet.h>                                           p.440
                    返回值：以主机字节序表示的16位整型数

int       open(const char *pathname, int oflag, ... /* mode_t mode */ );
                    <fcntl.h>                                               p.48
                    oflag: O_RDONLY, O_WRONLY, O_RDWR;
                           O_APPEND, O_CREAT, O_DSYNC, O_EXCL, O_NOCTTY,
                           O_NONBLOCK, O_RSYNC, O_SYNC, O_TRUNC
                    mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                    返回值：若成功则返回文件描述符，若出错则返回-1
                    平台：O_FSYNC标志在FreeBSD 5.2.1和Mac OS X 10.3上支持
```

822

```
DIR      *opendir(const char *pathname);
                   <direct.h>                                               p.98
```
返回值：若成功则返回指针，若出错则返回NULL

```
void     openlog(char *ident, int option, int facility);
                   <syslog.h>                                               p.346
                   option: LOG_CONS, LOG_NDELAY, LOG_NOWAIT, LOG_ODELAY,
                           LOG_PERROR, LOG_PID
                   facility: LOG_AUTH, LOG_AUTHPRIV, LOG_CRON, LOG_DAEMON,
                           LOG_FTP, LOG_KERN, LOG_LOCAL[0~7], LOG_LPR,
                           LOG_MAIL, LOG_NEWS, LOG_SYSLOG, LOG_USER, LOG_UUCP

long     pathconf(const char *pathname, int name);
                   <unistd.h>                                               p.33
                   name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,
                         _PC_FILESIZEBITS, _PC_LINK_MAX, _PC_MAX_CANON,
                         _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
                         _PC_PATH_MAX, _PC_PIPE_BUF, _PC_PRIO_IO,
                         _PC_SYMLINK_MAX, _PC_SYNC_IO, _PC_VDISABLE
```
返回值：若成功则返回相应值，若出错则返回−1

```
int      pause(void);
                   <unistd.h>                                               p.252
```
返回值：−1，并将errno设置为EINTR

```
int      pclose(FILE *fp);
                   <stdio.h>                                                p.403
```
返回值：popen *cmdstring*的终止状态，若出错则返回−1

```
void     perror(const char *msg);
                   <stdio.h>                                                p.11

int      pipe(int filedes[2]);
                   <unistd.h>                                               p.398
```
返回值：若成功则返回0，若出错则返回−1

```
int      poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
                   <poll.h>                                                 p.384
```
返回值：准备就绪的描述符数，若超时则返回0，若出错则返回−1
平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

```
FILE     *popen(const char *cmdstring, const char *type);
                   <stdio.h>                                                p.403
                   type: "r", "w"
```
返回值：若成功则返回文件指针，若出错则返回NULL

```
int      posix_openpt(int oflag);
                   <stdlib.h>                                               p.545
                   <fcntl.h>
                   oflag: O_RWDR, O_NOCTTY
```
返回值：若成功则返回下一个可用的PTY主设备的文件描述符，若出错则返回−1
平台：FreeBSD 5.2.1

823

```
ssize_t  pread(int filedes, void *buf, size_t nbytes, off_t offset);
                   <unistd.h>                                               p.59
```
返回值：读到的字节数，若已到文件结尾则返回0，若出错则返回−1

```
int      printf(const char *restrict format, ...);
                   <stdio.h>                                                p.121
```
返回值：若成功则返回输出字符数，若输出出错则返回负值

```
int     pselect(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writefds,
                fd_set *restrict exceptfds, const struct timespec *restrict tsptr,
                const sigset_t *restrict sigmask);
            <sys/select.h>                                                    p.384
```
返回值：准备就绪的描述符数，若超时则返回0，若出错则返回-1
平台：FreeBSD 5.2.1，Linux 2.4.22，Mac OS X 10.3

```
void    psignal(int signo, const char *msg);
            <signal.h>                                                        p.284
            <siginfo.h>     /* on Solaris */

int     pthread_atfork(void (*prepare)(void), void (*parent)(void),
                       void (*child)(void));
            <pthread.h>                                                       p.337
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_destroy(pthread_attr_t *attr);
            <pthread.h>                                                       p.314
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                    int *detachstate);
            <pthread.h>                                                       p.315
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                                  size_t *restrict guardsize);
            <pthread.h>                                                       p.317
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_getstack(const pthread_attr_t *restrict attr, void
                              **restrict stackaddr, size_t *restrict stacksize);
            <pthread.h>                                                       p.316
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                                  size_t *restrict stacksize);
            <pthread.h>                                                       p.317
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_init(pthread_attr_t *attr);
            <pthread.h>                                                       p.314
```
返回值：若成功则返回0，否则返回错误编号

824

```
int     pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
            <pthread.h>                                                       p.315
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
            <pthread.h>                                                       p.317
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_setstack(const pthread_attr_t *attr, void *stackaddr,
                              size_t *stacksize);
            <pthread.h>                                                       p.316
```
返回值：若成功则返回0，否则返回错误编号

```
int     pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
            <pthread.h>                                                       p.317
```
返回值：若成功则返回0，否则返回错误编号

int **pthread_cancel**(pthread_t *tid*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.295

void **pthread_cleanup_pop**(int *execute*);
<pthread.h>
p.295

void **pthread_cleanup_push**(void (**rtn*)(void *), void **arg*);
<pthread.h>
p.295

int **pthread_condattr_destroy**(pthread_condattr_t **attr*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.324

int **pthread_condattr_getpshared**(const pthread_condattr_t *restrict *attr*,
int *restrict *pshared*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.324

int **pthread_condattr_init**(pthread_condattr_t **attr*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.324

int **pthread_condattr_setpshared**(pthread_condattr_t **attr*, int *pshared*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.324

int **pthread_cond_broadcast**(pthread_cond_t **cond*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.310

825

int **pthread_cond_destroy**(pthread_cond_t **cond*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.309

int **pthread_cond_init**(pthread_cond_t *restrict *cond*,
pthread_condattr_t *restrict *attr*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.309

int **pthread_cond_signal**(pthread_cond_t **cond*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.310

int **pthread_cond_timedwait**(pthread_cond_t *restrict *cond*,
pthread_mutex_t *restrict *mutex*,
const struct timespec *restrict *timeout*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.309

int **pthread_cond_wait**(pthread_cond_t *restrict *cond*,
pthread_mutex_t *restrict *mutex*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.309

int **pthread_create**(pthread_t *restrict *tidp*,
const pthread_attr_t *restrict *attr*,
void *(**start_rtn*)(void), void *restrict *arg*);
<pthread.h>
返回值：若成功则返回0，否则返回错误编号
p.289

int **pthread_detach**(pthread_t *tid*);
<pthread.h> p.297
返回值：若成功则返回0，否则返回错误编号

int **pthread_equal**(pthread_t *tid1*, pthread_t *tid2*);
<pthread.h> p.288
返回值：若相等则返回非0值，否则返回0

void **pthread_exit**(void **rval_ptr*);
<pthread.h> p.291

int **pthread_getconcurrency**(void);
<pthread.h> p.317
返回值：当前的并发度

void ***pthread_getspecific**(pthread_key_t *key*);
<pthread.h> p.330
返回值：线程私有数据值，若没有值与键关联则返回NULL

int **pthread_join**(pthread_t *thread*, void ***rval_ptr*);
<pthread.h> p.291
返回值：若成功则返回0，否则返回错误编号

826

int **pthread_key_create**(pthread_key_t **keyp*, void (**destructor*)(void *);
<pthread.h> p.328
返回值：若成功则返回0，否则返回错误编号

int **pthread_key_delete**(pthread_key_t **key*);
<pthread.h> p.329
返回值：若成功则返回0，否则返回错误编号

int **pthread_kill**(pthread_t *thread*, int *signo*);
<signal.h> p.334
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutexattr_destroy**(pthread_mutexattr_t **attr*);
<pthread.h> p.318
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutexattr_getpshared**(const pthread_mutexattr_t *restrict *attr*,
int *restrict *pshared*);
<pthread.h> p.318
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutexattr_gettype**(const pthread_mutexattr_t *restrict *attr*,
int *restrict *type*);
<pthread.h> p.319
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutexattr_init**(pthread_mutexattr_t **attr*);
<pthread.h> p.318
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutexattr_setpshared**(pthread_mutexattr_t **attr*, int *pshared*);
<pthread.h> p.318
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutexattr_settype**(pthread_mutexattr_t **attr*, int *type*);
<pthread.h> p.319
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutex_destroy**(pthread_mutex_t **mutex*);
<pthread.h> p.299
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutex_init**(pthread_mutex_t *restrict *mutex*,
const pthread_mutexattr_t *restrict *attr*);
<pthread.h> p.299
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutex_lock**(pthread_mutex_t **mutex*);
<pthread.h> p.300
返回值：若成功则返回0，否则返回错误编号

827

int **pthread_mutex_trylock**(pthread_mutex_t **mutex*);
<pthread.h> p.300
返回值：若成功则返回0，否则返回错误编号

int **pthread_mutex_unlock**(pthread_mutex_t **mutex*);
<pthread.h> p.300
返回值：若成功则返回0，否则返回错误编号

int **pthread_once**(pthread_once_t **initflag*, void (**initfn*)(void);
<pthread.h> p.329
pthread_once_t *initflag* = PTHREAD_ONCE_INIT;
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlockattr_destroy**(pthread_rwlockattr_t **attr*);
<pthread.h> p.323
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlockattr_getpshared**(const pthread_rwlockattr_t *restrict *attr*,
int *restrict *pshared*);
<pthread.h> p.323
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlockattr_init**(pthread_rwlockattr_t **attr*);
<pthread.h> p.323
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlockattr_setpshared**(pthread_rwlockattr_t **attr*, int *pshared*);
<pthread.h> p.323
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlock_destroy**(pthread_rwlock_t **rwlock*);
<pthread.h> p.306
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlock_init**(pthread_rwlock_t *restrict *rwlock*,
const pthread_rwlockattr_t *restrict *attr*);
<pthread.h> p.306
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlock_rdlock**(pthread_rwlock_t **rwlock*);
<pthread.h> p.306
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlock_tryrdlock**(pthread_rwlock_t **rwlock*);
<pthread.h> p.306
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlock_trywrlock**(pthread_rwlock_t **rwlock*);
<pthread.h> p.306
返回值：若成功则返回0，否则返回错误编号

828

int **pthread_rwlock_unlock**(pthread_rwlock_t **rwlock*);
<pthread.h> p.306
返回值：若成功则返回0，否则返回错误编号

int **pthread_rwlock_wrlock**(pthread_rwlock_t **rwlock*);
<pthread.h> p.306
返回值：若成功则返回0，否则返回错误编号

pthread_t **pthread_self**(void);
<pthread.h> p.288
返回值：调用线程的线程ID

int **pthread_setcancelstate**(int *state*, int **oldstate*);
<pthread.h> p.331
返回值：若成功则返回0，否则返回错误编号

int **pthread_setcanceltype**(int *type*, int **oldtype*);
<pthread.h> p.332
返回值：若成功则返回0，否则返回错误编号

int **pthread_setconcurrency**(int *level*);
<pthread.h> p.317
返回值：若成功则返回0，否则返回错误编号

int **pthread_setspecific**(pthread_key_t *key*, const void **value*);
<pthread.h> p.330
返回值：若成功则返回0，否则返回错误编号

int **pthread_sigmask**(int *how*, const sigset_t *restrict *set*,
sigset_t *restrict *oset*);
<signal.h> p.334
返回值：若成功则返回0，否则返回错误编号

void **pthread_testcancel**(void);
<pthread.h> p.332

char ***ptsname**(int *filedes*);
<stdlib.h> p.546
返回值：若成功则返回指向PTY从设备名的指针，若出错则返回NULL
平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

int **putc**(int *c*, FILE **fp*);
<stdio.h> p.116
返回值：若成功则返回*c*，若出错则返回EOF

int **putchar**(int *c*);
<stdio.h> p.116
返回值：若成功则返回*c*，若出错则返回EOF

829

```
int       putchar_unlocked(int c);
                    <stdio.h>                                          p.326
                    返回值：若成功则返回c，若出错则返回EOF

int       putc_unlocked(int c, FILE *fp);
                    <stdio.h>                                          p.326
                    返回值：若成功则返回c，若出错则返回EOF

int       putenv(char *str);
                    <stdlib.h>                                         p.158
                    返回值：若成功则返回0，若出错则返回非0值

int       putmsg(int filedes, const struct strbuf *ctlptr,
                const struct strbuf *dataptr, int flag);
                    <stropts.h>                                        p.372
                    flag: 0, RS_HIPRI
                    返回值：若成功则返回0，若出错则返回-1
                    平台：Linux 2.4.22，Solaris 9

int       putpmsg(int filedes, const struct strbuf *ctlptr,
                const struct strbuf *dataptr, int band, int flag);
                    <stropts.h>                                        p.372
                    flag: 0, MSG_HIPRI, MSG_BAND
                    返回值：若成功则返回0，若出错则返回-1
                    平台：Linux 2.4.22，Solaris 9

int       puts(const char *str);
                    <stdio.h>                                          p.117
                    返回值：若成功则返回非负值，若出错则返回EOF

ssize_t   pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
                    <unistd.h>                                         p.59
                    返回值：若成功则返回已写的字节数，若出错则返回-1

int       raise(int signo);
                    <signal.h>                                         p.251
                    返回值：若成功则返回0，若出错则返回-1

ssize_t   read(int filedes, void *buf, size_t nbytes);
                    <unistd.h>                                         p.53
                    返回值：若成功则返回读到的字节数，若已到文件结尾则返回0，若出错则返回-1

struct
dirent    *readdir(DIR *dp);
                    <dirent.h>                                         p.98
                    返回值：若成功则返回指针，若在目录结尾或出错则返回NULL

int       readlink(const char *restrict pathname, char *restrict buf,
                size_t bufsize);
                    <unistd.h>                                         p.94
                    返回值：若成功则返回读到的字节数，若出错则返回-1

ssize_t   readv(int filedes, const struct iovec *iov, int iovcnt);
                    <sys/uio.h>                                        p.387
                    返回值：若成功则返回已读的字节数，若出错则返回-1

void      *realloc(void *ptr, size_t newsize);
                    <stdlib.h>                                         p.155
                    返回值：若成功则返回非空指针，若出错则返回NULL
```

830

ssize_t **recv**(int *sockfd*, void **buf*, size_t *nbytes*, int *flags*);

<sys/socket.h> p.454

flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL

返回值：以字节计数的消息长度，若无可用消息或对方已经按序结束则返回0，若出错则返回−1

平台：MSG_TRUNC标志在Linux 2.4.22上支持

ssize_t **recvfrom**(int *sockfd*, void *restrict *buf*, size_t *len*, int *flags*, struct sockaddr *restrict *addr*, socklen_t *restrict *addrlen*);

<sys/socket.h> p.455

flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL

返回值：以字节计数的消息长度，若无可用消息或对方已经按序结束则返回0，若出错则返回−1

平台：MSG_TRUNC标志在Linux 2.4.22上支持

ssize_t **recvmsg**(int *sockfd*, struct msghdr **msg*, int *flags*);

<sys/socket.h> p.455

flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL

返回值：以字节计数的消息长度，若无可用消息或对方已经按序结束则返回0，若出错则返回−1

平台：MSG_TRUNC标志在Linux 2.4.22上支持

int **remove**(const char **pathname*);

<stdio.h> p.90

返回值：若成功则返回0，若出错则返回−1

int **rename**(const char **oldname*, const char **newname*);

<stdio.h> p.91

返回值：若成功则返回0，若出错则返回−1

void **rewind**(FILE **fp*);

<stdio.h> p.120

void **rewinddir**(DIR **dp*);

<dirent.h> p.98

int **rmdir**(const char **pathname*);

<unistd.h> p.98

返回值：若成功则返回0，若出错则返回−1

int **scanf**(const char *restrict *format*, ...);

<stdio.h> p.124

返回值：指定的输入项数，若输入出错或在任意变换前已到达文件结尾则返回EOF

831

void **seekdir**(DIR **dp*, long *loc*);

<dirent.h> p.98

int **select**(int *maxfdp1*, fd_set *restrict *readfds*, fd_set *restrict *writefds*, fd_set *restrict *exceptfds*, struct timeval *restrict *tvptr*);

<sys/select.h> p.381

返回值：准备就绪的描述符数，若超时则返回0，若出错则返回−1

int **semctl**(int *semid*, int *semnum*, int *cmd*, ... /* union semun *arg* */);

<sys/sem.h> p.424

cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT, GETZCNT, GETVAL, SETVAL, GETALL, SETALL

返回值：(和命令相关)

```
int       semget(key_t key, int nsems, int flag);
                    <sys/sem.h>                                                     p.423
                    flag: 0, IPC_CREAT, IPC_EXCL
                    返回值：若成功则返回信号量ID，若出错则返回-1

int       semop(int semid, struct sembuf semoparray[], size_t nops);
                    <sys/sem.h>                                                     p.425
                    返回值：若成功则返回0，若出错则返回-1

ssize_t   send(int sockfd, const void *buf, size_t nbytes, int flags);
                    <sys/socket.h>                                                  p.453
                    flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
                    返回值：若成功则返回发送的字节数，若出错则返回-1
                    平台：MSG_DONTWAIT标志在FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3上支持，
                         MSG_EOR标志在Solaris 9上不支持

ssize_t   sendmsg(int sockfd, const struct msghdr *msg, int flags);
                    <sys/socket.h>                                                  p.454
                    flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
                    返回值：若成功则返回发送的字节数，若出错则返回-1
                    平台：MSG_DONTWAIT标志在FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3上支持，
                         MSG_EOR标志在Solaris 9上不支持

ssize_t   sendto(int sockfd, const void *buf, size_t nbytes, int flags,
                 const struct sockaddr *destaddr, socklen_t destlen);
                    <sys/socket.h>                                                  p.453
                    flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
                    返回值：若成功则返回发送的字节数，若出错则返回-1
                    平台：MSG_DONTWAIT标志在FreeBSD 5.2.1、Linux 2.4.22和Mac OS X 10.3上支持，
                         MSG_EOR标志在Solaris 9上不支持

void      setbuf(FILE *restrict fp, char *restrict buf);
                    <stdio.h>                                                       p.111

int       setegid(gid_t gid);
                    <unistd.h>                                                      p.195
                    返回值：若成功则返回0，若出错则返回-1

int       setenv(const char *name, const char *value, int rewrite);
                    <stdlib.h>                                                      p.158
                    返回值：若成功则返回0，若出错则返回非0值

int       seteuid(uid_t uid);
                    <unistd.h>                                                      p.195
                    返回值：若成功则返回0，若出错则返回-1

int       setgid(gid_t gid);
                    <unistd.h>                                                      p.193
                    返回值：若成功则返回0，若出错则返回-1

void      setgrent(void);
                    <grp.h>                                                         p.137

int       setgroups(int ngroups, const gid_t grouplist[]);
                    <grp.h>    /* on Linux */                                       p.138
                    <unistd.h> /* on FreeBSD, Mac OS X, and Solaris */
                    返回值：若成功则返回0，若出错则返回-1

void      sethostent(int stayopen);
                    <netdb.h>                                                       p.443
```

832

int **setjmp**(jmp_buf *env*);
<setjmp.h> p.161
返回值：若直接调用则返回0，若从longjmp调用返回则返回非0值

int **setlogmask**(int *maskpri*);
<syslog.h> p.346
返回值：前日志记录优先级屏蔽值

void **setnetent**(int *stayopen*);
<netdb.h> p.443

int **setpgid**(pid_t *pid*, pid_t *pgid*);
<unistd.h> p.218
返回值：若成功则返回0，若出错则返回−1

void **setprotoent**(int *stayopen*);
<netdb.h> p.444

void **setpwent**(void);
<pwd.h> p.135

int **setregid**(gid_t *rgid*, gid_t *egid*);
<unistd.h> p.195
返回值：若成功则返回0，若出错则返回−1

int **setreuid**(uid_t *ruid*, uid_t *euid*);
<unistd.h> p.195
返回值：若成功则返回0，若出错则返回−1

833

int **setrlimit**(int *resource*, const struct rlimit **rlptr*);
<sys/resource.h> p.165
返回值：若成功则返回0，若出错则返回非0值

void **setservent**(int *stayopen*);
<netdb.h> p.444

pid_t **setsid**(void);
<unistd.h> p.219
返回值：若成功则返回进程组ID，若出错则返回−1

int **setsockopt**(int *sockfd*, int *level*, int *option*, const void **val*, socklen_t *len*);
<sys/socket.h> p.464
返回值：若成功则返回0，若出错则返回−1

void **setspent**(void);
<shadow.h> p.137
平台：Linux 2.4.22，Solaris 9

int **setuid**(uid_t *uid*);
<unistd.h> p.193
返回值：若成功则返回0，若出错则返回−1

int **setvbuf**(FILE *restrict *fp*, char *restrict *buf*, int *mode*, size_t *size*);
<stdio.h> p.111
mode：_IOFBF，_IOLBF，_IONBF
返回值：若成功则返回0，若出错则返回非0值

```
void    *shmat(int shmid, const void *addr, int flag);
                <sys/shm.h>                                              p.428
                flag: 0, SHM_RND, SHM_RDONLY
                返回值：若成功则返回指向共享存储的指针，若出错则返回-1

int     shmctl(int shmid, int cmd, struct shmid_ds *buf);
                <sys/shm.h>                                              p.428
                cmd: IPC_STAT, IPC_SET, IPC_RMID,
                     SHM_LOCK, SHM_UNLOCK
                返回值：若成功则返回0，若出错则返回-1

int     shmdt(void *addr);
                <sys/shm.h>                                              p.429
                返回值：若成功则返回0，若出错则返回-1

int     shmget(key_t key, int size, int flag);
                <sys/shm.h>                                              p.427
                flag: 0, IPC_CREAT, IPC_EXCL
                返回值：若成功则返回共享存储ID，若出错则返回-1

int     shutdown(int sockfd, int how);
                <sys/socket.h>                                           p.439
                how: SHUT_RD, SHUT_WR, SHUT_RDWR
                返回值：若成功则返回0，若出错则返回-1

int     sig2str(int signo, char *str);
                <signal.h>                                               p.285
                返回值：若成功则返回0，若出错则返回-1
                平台：Solaris 9

int     sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
                <signal.h>                                               p.261
                返回值：若成功则返回0，若出错则返回-1

int     sigaddset(sigset_t *set, int signo);
                <signal.h>                                               p.257
                返回值：若成功则返回0，若出错则返回-1

int     sigdelset(sigset_t *set, int signo);
                <signal.h>                                               p.257
                返回值：若成功则返回0，若出错则返回-1

int     sigemptyset(sigset_t *set);
                <signal.h>                                               p.257
                返回值：若成功则返回0，若出错则返回-1

int     sigfillset(sigset_t *set);
                <signal.h>                                               p.257
                返回值：若成功则返回0，若出错则返回-1

int     sigismember(const sigset_t *set, int signo);
                <signal.h>                                               p.257
                返回值：若真则返回1，若假则返回0，若出错则返回-1
```

834

```
void      siglongjmp(sigjmp_buf env, int val);
                    <setjmp.h>                                               p.266
                    此函数不返回

void    (*signal(int signo, void (*func)(int)))(int);
                    <signal.h>                                               p.240
                    返回值：若成功则返回信号以前的处理配置，若出错则返回SIG_ERR

int       sigpending(sigset_t *set);
                    <signal.h>                                               p.259
                    返回值：若成功则返回0，若出错则返回-1
```

835

```
int       sigprocmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
                    <signal.h>                                               p.258
                    how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
                    返回值：若成功则返回0，若出错则返回-1

int       sigsetjmp(sigjmp_buf env, int savemask);
                    <setjmp.h>                                               p.266
                    返回值：若直接调用则返回0，若从siglongjmp调用返回则返回非0值

int       sigsuspend(const sigset_t *sigmask);
                    <signal.h>                                               p.269
                    返回值：-1，并将errno设置为EINTR

int       sigwait(const sigset_t *restrict set, int *restrict signop);
                    <signal.h>                                               p.334
                    返回值：若成功则返回0，否则返回错误编号
unsigned
int       sleep(unsigned int seconds);
                    <unistd.h>                                               p.280
                    返回值：0或未休眠够的秒数

int       snprintf(char *restrict buf, size_t n, const char *restrict format, ...);
                    <stdio.h>                                                p.121
                    返回值：若成功则返回存入数组的字符数，若编码出错则返回负值

int       sockatmark(int sockfd);
                    <sys/socket.h>                                           p.467
                    返回值：若在标记处则返回1，若没有在标记处则返回0，若出错则返回-1

int       socket(int domain, int type, int protocol);
                    <sys/socket.h>                                           p.437
                    type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,
                    返回值：若成功则返回文件（套接字）描述符，若出错则返回-1

int       socketpair(int domain, int type, int protocol, int sockfd[2]);
                    <sys/socket.h>                                           p.476
                    type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,
                    返回值：若成功则返回0，若出错则返回-1

int       sprintf(char *restrict buf, const char *restrict format, ...);
                    <stdio.h>                                                p.121
                    返回值：若成功则返回存入数组的字符数，若编码出错则返回负值
```

836

int **sscanf**(const char *restrict *buf*, const char *restrict *format*, ...);
<stdio.h> p.124
返回值：指定的输入项数，若输入出错或在任意变换前已到达文件结尾则返回EOF

int **stat**(const char *restrict *pathname*, struct stat *restrict *buf*);
<sys/stat.h> p.71
返回值：若成功则返回0，若出错则返回−1

int **str2sig**(const char *str*, int *signop*);
<signal.h> p.285
返回值：若成功则返回0，若出错则返回−1
平台：Solaris 9

char ***strerror**(int *errnum*);
<string.h> p.11
返回值：指向消息字符串的指针

size_t **strftime**(char *restrict *buf*, size_t *maxsize*, const char *restrict *format*,
const struct tm *restrict *tmptr*);
<time.h> p.144
返回值：若有空间则返回存入数组的字符数，否则返回0

char ***strsignal**(int *signo*);
<string.h> p.284
返回值：指向描述该信号的字符串的指针

int **symlink**(const char *actualpath*, const char *sympath*);
<unistd.h> p.94
返回值：若成功则返回0，若出错则返回−1

void **sync**(void);
<unistd.h> p.62

long **sysconf**(int *name*);
<unistd.h> p.33
name: _SC_ARG_MAX, _SC_ATEXIT_MAX, _SC_CHILD_MAX,
_SC_CLK_TCK, _SC_COLL_WEIGHTS_MAX,
_SC_HOST_NAME_MAX, _SC_IOV_MAX, _SC_JOB_CONTROL,
_SC_LINE_MAX, _SC_LOGIN_NAME_MAX, _SC_NGROUPS_MAX,
_SC_OPEN_MAX, _SC_PAGESIZE, _SC_PAGE_SIZE,
_SC_READER_WRITER_LOCKS, _SC_RE_DUP_MAX,
_SC_SAVED_IDS, _SC_SHELL, _SC_STREAM_MAX,
_SC_SYMLOOP_MAX, _SC_TTY_NAME_MAX, _SC_TZNAME_MAX,
_SC_VERSION, _SC_XOPEN_CRYPT, _SC_XOPEN_LEGACY,
_SC_XOPEN_REALTIME, _SC_XOPEN_REALTIME_THREADS,
_SC_XOPEN_VERSION
返回值：若成功则返回相应值，若出错则返回−1

void **syslog**(int *priority*, char *format*, ...);
<syslog.h> p.346

int **system**(const char *cmdstring*);
<stdlib.h> p.200
返回值：shell的终止状态

837

int **tcdrain**(int *filedes*);
<termios.h> p.524
返回值：若成功则返回0，若出错则返回−1

```
int      tcflow(int filedes, int action);
                   <termios.h>                                                p.524
                   action: TCOOFF, TCOON, TCIOFF, TCION
                   返回值：若成功则返回0，若出错则返回-1

int      tcflush(int filedes, int queue);
                   <termios.h>                                                p.524
                   queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
                   返回值：若成功则返回0，若出错则返回-1

int      tcgetattr(int filedes, struct termios *termptr);
                   <termios.h>                                                p.516
                   返回值：若成功则返回0，若出错则返回-1

pid_t    tcgetpgrp(int filedes);
                   <unistd.h>                                                 p.221
                   返回值：若成功则返回前台进程组的进程组ID，若出错则返回-1

pid_t    tcgetsid(int filedes);
                   <termios.h>                                                p.222
                   返回值：若成功则返回会话首进程的进程组ID，若出错则返回-1

int      tcsendbreak(int filedes, int duration);
                   <termios.h>                                                p.524
                   返回值：若成功则返回0，若出错则返回-1

int      tcsetattr(int filedes, int opt, const struct termios *termptr);
                   <termios.h>                                                p.516
                   opt: TCSANOW, TCSADRAIN, TCSAFLUSH
                   返回值：若成功则返回0，若出错则返回-1

int      tcsetpgrp(int filedes, pid_t pgrpid);
                   <unistd.h>                                                 p.221
                   返回值：若成功则返回0，若出错则返回-1

long     telldir(DIR *dp);
                   <dirent.h>                                                 p.98
                   返回值：与dp关联的目录中的当前位置

char     *tempnam(const char *directory, const char *prefix);
                   <stdio.h>                                                  p.128
                   返回值：指向唯一路径名的指针

time_t   time(time_t *calptr);
                   <time.h>                                                   p.142
                   返回值：若成功则返回时间值，若出错则返回-1

clock_t  times(struct tms *buf);
                   <sys/times.h>                                              p.208
                   返回值：若成功则返回流逝的墙上时钟时间（单位：时钟滴答数），若出错则返回-1

FILE     *tmpfile(void);
                   <stdio.h>                                                  p.127
                   返回值：若成功则返回文件指针，若出错则返回NULL

char     *tmpnam(char *ptr);
                   <stdio.h>                                                  p.127
                   返回值：指向唯一路径名的指针
```

838

int **truncate**(const char *pathname, off_t length);
<unistd.h> p.86
返回值：若成功则返回0，若出错则返回−1

char ***ttyname**(int filedes);
<unistd.h> p.525
返回值：指向终端路径名的指针，若出错则返回NULL

mode_t **umask**(mode_t cmask);
<sys/stat.h> p.79
返回值：以前的文件模式创建屏蔽字

int **uname**(struct utsname *name);
<sys/utsname.h> p.141
返回值：若成功则返回非负值，若出错则返回−1

int **ungetc**(int c, FILE *fp);
<stdio.h> p.115
返回值：若成功则返回c，若出错则返回EOF

int **unlink**(const char *pathname);
<unistd.h> p.89
返回值：若成功则返回0，若出错则返回−1

int **unlockpt**(int filedes);
<stdlib.h> p.545
返回值：若成功则返回0，若出错则返回−1
平台：FreeBSD 5.2.1，Linux 2.4.22，Solaris 9

void **unsetenv**(const char *name);
<stdlib.h> p.158

int **utime**(const char *pathname, const struct utimbuf *times);
<utime.h> p.96
返回值：若成功则返回0，若出错则返回−1

839

int **vfprintf**(FILE *restrict fp, const char *restrict format, va_list arg);
<stdarg.h> p.123
<stdio.h>
返回值：若成功则返回输出字符数，若输出出错则返回负值

int **vfscanf**(FILE *restrict fp, const char *restrict format, va_list arg);
<stdarg.h> p.125
<stdio.h>
返回值：指定的输入项数，若输入出错或在任一变换前已到达文件结尾则返回EOF

int **vprintf**(const char *restrict format, va_list arg);
<stdarg.h> p.123
<stdio.h>
返回值：若成功则返回输出字符数，若输出出错则返回负值

int **vscanf**(const char *restrict format, va_list arg);
<stdarg.h> p.125
<stdio.h>
返回值：指定的输入项数，若输入出错或在任一变换前已到达文件结尾则返回EOF

int **vsnprintf**(char *restrict *buf*, size_t *n*, const char *restrict *format*,
va_list *arg*);

<stdarg.h> p.123
<stdio.h>

返回值：若成功则返回存入数组的字符数，若编码出错则返回负值

int **vsprintf**(char *restrict *buf*, const char *restrict *format*, va_list *arg*);

<stdarg.h> p.123
<stdio.h>

返回值：若成功则返回存入数组的字符数，若编码出错则返回负值

int **vsscanf**(const char *restrict *buf*, const char *restrict *format*,
va_list *arg*);

<stdarg.h> p.125
<stdio.h>

返回值：指定的输入项数，若输入出错或在任一变换前已到达文件结尾则返回EOF

void **vsyslog**(int *priority*, const char **format*, va_list *arg*);

<syslog.h> p.348
<stdarg.h>

pid_t **wait**(int **statloc*);

<sys/wait.h> p.179

返回值：若成功则返回进程ID，0，若出错则返回-1

840

int **waitid**(idtype_t *idtype*, id_t *id*, siginfo_t **infop*, int *options*);

<sys/wait.h> p.184

idtype: P_PID, P_PGID, P_ALL

options: WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED

返回值：若成功则返回0，若出错则返回-1

平台：Solaris 9

pid_t **waitpid**(pid_t *pid*, int **statloc*, int *options*);

<sys/wait.h> p.179

options: 0, WCONTINUED, WNOHANG, WUNTRACED

返回值：若成功则返回进程ID，0，若出错则返回-1

pid_t **wait3**(int **statloc*, int *options*, struct rusage **rusage*);

<sys/types.h> p.184
<sys/wait.h>
<sys/time.h>
<sys/resource.h>

options: 0, WNOHANG, WUNTRACED

返回值：若成功则返回进程ID，0，若出错则返回-1

pid_t **wait4**(pid_t *pid*, int **statloc*, int *options*, struct rusage **rusage*);

<sys/types.h> p.184
<sys/wait.h>
<sys/time.h>
<sys/resource.h>

options: 0, WNOHANG, WUNTRACED

返回值：若成功则返回进程ID，0，若出错则返回-1

ssize_t **write**(int *filedes*, const void **buf*, size_t *nbytes*);

<unistd.h> p.54

返回值：若成功则返回已写的字节数，若出错则返回-1

ssize_t **writev**(int *filedes*, const struct iovec **iov*, int *iovcnt*);

<sys/uio.h> p.387

返回值：若成功则返回已写的字节数，若出错则返回-1

841

附录B

其他源代码

B.1 本书使用的头文件

正文中的大多数程序都包含头文件apue.h，见程序清单B-1。其中定义了常量（如MAXLINE）和我们自编的函数的原型。

因为大多数程序需要包含下列头文件：<stdio.h>、<stdlib.h>（其中有exit函数原型）和<unistd.h>（其中包含所有标准UNIX函数原型），所以apue.h自动包含了这些系统头文件，同时还包含了<string.h>。这样就减少了本书正文中列出的所有程序的长度。

程序清单B-1　头文件apue.h

```
/* Our own header, to be included before all standard system headers */

#ifndef _APUE_H
#define _APUE_H

#define _XOPEN_SOURCE   600 /* Single UNIX Specification, Version 3 */

#include <sys/types.h>      /* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h>    /* for winsize */
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif
#include <stdio.h>      /* for convenience */
#include <stdlib.h>     /* for convenience */
#include <stddef.h>     /* for offsetof */
#include <string.h>     /* for convenience */
#include <unistd.h>     /* for convenience */
#include <signal.h>     /* for SIG_ERR */
#define MAXLINE 4096            /* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void    Sigfunc(int);   /* for signal handlers */

#if defined(SIG_IGN) && !defined(SIG_ERR)
#define SIG_ERR ((Sigfunc *)-1)
```

843

```
#endif

#define min(a,b)     ((a) < (b) ? (a) : (b))
#define max(a,b)     ((a) > (b) ? (a) : (b))

/*
 * Prototypes for our own functions.
 */
char    *path_alloc(int *);
long     open_max(void);
void     clr_fl(int, int);
void     set_fl(int, int);
void     pr_exit(int);
void     pr_mask(const char *);
Sigfunc *signal_intr(int, Sigfunc *);

int      tty_cbreak(int);
int      tty_raw(int);
int      tty_reset(int);
void     tty_atexit(void);
#ifdef  ECHO    /* only if <termios.h> has been included */
struct termios  *tty_termios(void);
#endif

void     sleep_us(unsigned int);
ssize_t  readn(int, void *, size_t);
ssize_t  writen(int, const void *, size_t);
void     daemonize(const char *);

int      s_pipe(int *);
int      recv_fd(int, ssize_t (*func)(int,
                 const void *, size_t));
int      send_fd(int, int);
int      send_err(int, int,
                  const char *);
int      serv_listen(const char *);
int      serv_accept(int, uid_t *);
int      cli_conn(const char *);
int      buf_args(char *, int (*func)(int,
                  char **));

int      ptym_open(char *, int);
int      ptys_open(char *);
#ifdef  TIOCGWINSZ
pid_t    pty_fork(int *, char *, int, const struct termios *,
                  const struct winsize *);
#endif

int     lock_reg(int, int, int, off_t, int, off_t);
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t   lock_test(int, int, off_t, int, off_t);

#define is_read_lockable(fd, offset, whence, len) \
```

844

```
            (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void    err_dump(const char *, ...);
void    err_msg(const char *, ...);
void    err_quit(const char *, ...);
void    err_exit(int, const char *, ...);
void    err_ret(const char *, ...);
void    err_sys(const char *, ...);

void    log_msg(const char *, ...);
void    log_open(const char *, int, int);
void    log_quit(const char *, ...);
void    log_ret(const char *, ...);
void    log_sys(const char *, ...);

void    TELL_WAIT(void);
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif  /* _APUE_H */
```

845

程序中先包括apue.h，然后再包括一般的系统头文件，这样做的原因是使我们可以先定义一些在此后包括的头文件可能要求的内容，控制头文件被包括的顺序，以及使我们可以重定义某些部分，而这正是为隐藏系统之间的差别而需要解决的。

B.2 标准出错处理例程

我们提供了两套出错处理函数，它们用于本书中大多数实例以处理各种出错情况。一套以err_开头，向标准出错文件输出一条出错消息；另一套以log_开头，用于多半没有控制终端的守护进程（见第13章）。

提供了这些出错处理函数后，只要在程序中写一行C代码就可以进行出错处理，例如：

```
if (出错条件)
    err_dump(带任意参数的printf格式);
```

这样也就不再需要使用下列代码：

```
if  (出错条件) {
     char  buf[200];
     sprintf(buf, 带任意参数的printf格式);
     perror(buf);
     abort( );
}
```

我们的出错处理函数使用了ISO C 的变长参数表设施。其详细说明见Kernighan和Ritchie[1998]的7.3节。应当注意的是这一ISO C 设施与早期系统（例如SVR3和4.3BSD）提供的varargs设施不同。虽然宏的名字相同，但宏的某些参数已经发生了改变。

表B-1列出了各个出错处理函数之间的区别。

程序清单B-2给出了输出至标准出错文件的出错处理函数。

表B-1　标准出错处理函数

函　数	从strerror增加字符串?	strerror的参数	终　止?
err_dump	是	errno	abort();
err_exit	是	显式参数	exit(1);
err_msg	否		return;
err_quit	否		exit(1);
err_ret	是	errno	return;
err_sys	是	errno	exit(1);
log_msg	否		return;
log_quit	否		exit(2);
log_ret	是	errno	return;
log_sys	是	errno	exit(2);

846

程序清单B-2　输出至标准出错文件的出错处理函数

```
#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable aruments */

static void err_doit(int, int, const char *, va_list);

/*
 * Nonfatal error related to a system call.
 * Print a message and return.
 */
void
err_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
err_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Fatal error unrelated to a system call.
 * Error code passed as explict parameter.
 * Print a message and terminate.
 */
void
err_exit(int error, const char *fmt, ...)
{
```

```
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
    exit(1);
}
```

847

```
/*
 * Fatal error related to a system call.
 * Print a message, dump core, and terminate.
 */
void
err_dump(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort();        /* dump core and terminate */
    exit(1);        /* shouldn't get here */
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
err_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
err_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag".
 */
static void
err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
    char    buf[MAXLINE];
    vsnprintf(buf, MAXLINE, fmt, ap);
```

848

```
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
          strerror(error));
    strcat(buf, "\n");
    fflush(stdout);     /* in case stdout and stderr are the same */
    fputs(buf, stderr);
    fflush(NULL);       /* flushes all stdio output streams */
}
```

程序清单B-3给出了各log_xxx出错处理函数。若进程不以守护进程方式运行，那么这些函数要求调用者定义变量log_to_stderr，并将其设置为非0值。在这种情况下，出错消息被送至标准出错文件。若log_to_stderr标志为0，则使用syslog设施（见13.4节）。

程序清单B-3　用于守护进程的出错处理函数

```
/*
 * Error routines for programs that can run as a daemon.
 */

#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable arguments */
#include <syslog.h>

static void log_doit(int, int, const char *, va_list ap);

/*
 * Caller must define and set this: nonzero if
 * interactive, zero if daemon
 */
extern int  log_to_stderr;

/*
 * Initialize syslog(), if running as daemon.
 */
void
log_open(const char *ident, int option, int facility)
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Nonfatal error related to a system call.
 * Print a message with the system's errno value and return.
 */
void
log_ret(const char *fmt, ...)
{
    va_list     ap;
    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
log_sys(const char *fmt, ...)
```

849

```
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
log_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
log_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag" and "priority".
 */
static void
log_doit(int errnoflag, int priority, const char *fmt, va_list ap)
{
    int     errno_save;
    char    buf[MAXLINE];

    errno_save = errno;     /* value caller might want printed */
    vsnprintf(buf, MAXLINE, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
          strerror(errno_save));
    strcat(buf, "\n");
    if (log_to_stderr) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else {
        syslog(priority, buf);
    }
}
```

850

851

附录C

部分习题答案

第1章

1.1 这个习题利用ls(1)命令的下面两个参数：-i，显示文件或目录的i节点号（4.14节中详细讨论i节点）；-d，显示目录的信息，而不是目录中所有文件的信息。

执行下列命令：

```
$ ls -ldi /etc/. /etc/..                    -i要求打印i节点号
 162561 drwxr-xr-x  66 root    4096 Feb  5 03:59 /etc/./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /etc/../
$ ls -ldi /. /..                            .和..的i节点号均为2
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /../
```

1.2 UNIX系统是多道程序或多任务系统，所以，在程序清单1-4中的程序运行的同时其他两个进程也在运行。

1.3 若perror的*ptr*参数是一个指针，则perror可以改变*ptr*所指字符串。利用限定符const使得perror不能修改*ptr*所指的字符串。而strerror的参数是错误号，它是整型的并且C按值传递所有参数，因此strerror函数即使想也不能修改这个值。（如果对C中函数参数的处理不是很清楚，可参见Kernighan和Ritchie[1988]的5.2节。）

853

1.4 调用fflush、fprintf和vprintf函数可修改errno的值。如果这些函数修改了errno的值但我们没有保存，则最终显示的出错消息是不正确的。

1.5 在2038年。将time_t数据类型定为64位整型，就可以解决该问题了。如果它现在是32位整型，那么为使应用程序正常工作应当对其进行重新编译。但是这一问题还有更糟糕之处。某些文件系统及备份介质以32位整型存储时间。对于这些同样需要加以更新，但又需能读旧的格式。

1.6 大约248天。

第2章

2.1 下面是FreeBSD中使用的技术。在头文件<machine/_types.h>中定义可在多个头文件中出现的基本数据类型。例如：

```
#ifndef _MACHINE__TYPES_H_
#define _MACHINE__TYPES_H_

typedef int               __int32_t;
typedef unsigned int      __uint32_t;
...
```

```
typedef __uint32_t      __size_t;
...

#endif  /* _MACHINE__TYPES_H_ */
```

在每个可以定义基本系统数据类型size_t的头文件中，我们可以有序列。

```
#ifndef  _SIZE_T_DECLARED
typedef __size_t       size_t;
#define _SIZE_T_DECLARED
#endif
```

这样，实际上只执行一次size_t的typedef。

2.3 如果OPEN_MAX是未确定的或大得出奇（即等于LONG_MAX），那么我们可以使用getrlimit以得到每个进程的最大打开文件描述符数。因为可以修改对每个进程的限制，所以我们不能将从前一个调用得到的值高速缓存起来（它可能已被更改），见程序清单C-1。

854

程序清单C-1　标识最大可能文件描述符的替换方法

```
#include "apue.h"
#include <limits.h>
#include <sys/resource.h>

#define OPEN_MAX_GUESS  256

long
open_max(void)
{
    long openmax;
    struct rlimit rl;

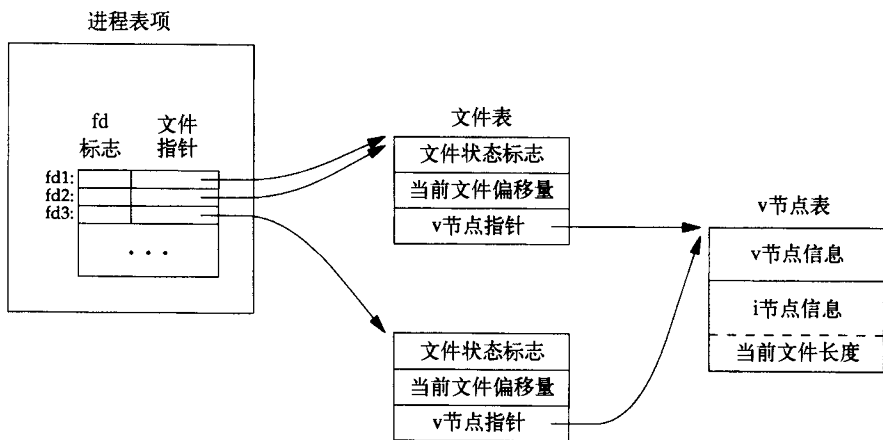
    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0 ||
      openmax == LONG_MAX) {
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
            err_sys("can't get file limit");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}
```

第3章

3.1 所有的磁盘I/O都要经过内核的块缓冲区（也称为内核的缓冲区高速缓存），唯一例外的是对原始磁盘设备的I/O，但是我们不考虑这种情况。Bach[1986]的第3章描述了这种缓冲区高速缓存的操作）。既然read或write的数据都要被内核缓冲，那么术语“不带缓冲的I/O”指的是在用户的进程中对这两个函数不会自动缓冲，每次read或write就要进行一次系统调用。

3.3 每次调用open函数就分配一个新的文件表项，但是因为两次打开的是相同的文件，所以两个文件表项指向相同的v节点。调用dup引用已存在的文件表项(此处指fd1的文件表项)，见图C-1。当F_SETFD作用于fd1时，只影响fd1的文件描述符标志；F_SETFL作用于

fd1时，则影响fd1及fd2指向的文件表项。



图C-1 open和dup的结果

3.4 如果fd是1，执行dup2(fd,1)后返回1，但是没有关闭描述符1（见3.12节讨论）。调用3次dup2后，3个描述符指向相同的文件表项，所以不需要关闭描述符。

但如果fd是3，调用3次dup2后，有4个描述符指向相同的文件表项，这种情况下就需要关闭描述符3。

855

3.5 因为shell从左到右处理命令行,所以

```
./a.out > outfile 2>&1
```

首先设置标准输出到outfile，然后执行dup将标准输出复制到描述符2（标准错误）上，其结果是将标准输出和标准错误设置为同一文件，即描述符1和描述符2指向同一个文件表项。而对于命令行

```
./a.out 2>&1 > outfile
```

由于首先执行dup，所以使描述符2成为终端（假设命令是交互运行的），标准输出重定向到outfile。结果是描述符1指向outfile的文件表项，描述符2指向终端的文件表项。

3.6 这种情况之下，仍然可以用lseek和read函数读文件中任意一处的内容。但是write函数在写数据之前会自动将文件偏移量设置为文件尾，所以写文件时只能从文件尾开始。

第4章

4.1 若调用stat函数，它总是顺着符号链接向前（表4-9），所以该程序决不会显示文件类型是“符号链接”。例如，正如本书文中所示/dev/cdrom是cdroms/cdrom0的一个符号链接（它本身也是到../scsi/host0/bus0/target0/lun0/cd的符号链接），但是stat函数的结果只显示/dev/cdrom是一个块特殊文件，而不说明它是一个符号链接。若符号链接指向一个不存在的文件，则stat返回出错。

856

4.2 关闭了该文件的所有访问权限：

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
----------  1 sar          0 Feb  5 14:06 temp.foo
```

4.3 下面的命令表示关闭用户读权限时所发生的情况：

```
$ date > foo
$ chmod u-r foo                          关闭用户读权限
$ ls -l foo                              验证文件的权限
--w-r--r--  1 sar         29 Feb  5 14:21 foo
$ cat foo                                读文件
cat: foo: Permission denied
```

4.4 如果用open或creat创建已经存在的文件，则该文件的访问权限位不变。运行程序清单4-3中的程序可以验证这点。

```
$ rm foo bar                         删除文件
$ date > foo                         创建文件
$ date > bar
$ chmod a-r foo bar                  关闭所有的读权限
$ ls -l foo bar                      验证其权限
--w-------  1 sar         29 Feb  5 14:25 bar
--w-------  1 sar         29 Feb  5 14:25 foo
$ ./a.out                            运行图4.9程序
$ ls -l foo bar                      检查文件的权限和大小
--w-------  1 sar          0 Feb  5 14:26 bar
--w-------  1 sar          0 Feb  5 14:26 foo
```

注意，访问权限没有改变，但是文件长度截短了。

4.5 目录的长度从来不会是0，因为它总是包含.和..两项。符号链接的长度指其路径名包含的字符数，由于路径名中至少有一个字符，所以长度也不为0。

4.7 当创建新的core文件时，内核对其访问权限有一个默认设置，在本例中是rw-r--r--。这一默认值可能会也可能不会被umask的值修改。shell对创建的重定向的新文件也有一个默认的访问权限，本例中为rw-rw-rw-，并且这个值总是被当前的umask修改，本例中umask为02。

4.8 不能使用du的原因是它需要文件名，如

```
du tempfile
```

或目录名，如

```
du .
```

857

只有当unlink函数返回时才释放tempfile的目录项，du.命令没有计算仍然被tempfile占用的空间。本例中只能使用df命令查看文件系统中实际可用的空闲空间。

4.9 如果被删除的链接不是该文件的最后一个链接，则不会删除该文件。此时，文件的状态改变时间被更新。但是，如果被删除的链接是最后一个链接，则该文件将被物理删除。这时再去更新文件的状态改变时间就没有意义，因为包含文件所有信息的i节点将会随着文件的删除而被释放。

4.10 用opendir打开一个目录后，递归调用函数dopath。假设opendir使用一个文件描述符，并且只有在处理完目录后才调用closedir释放描述符，这就意味着每次降一级就要使用另外一个描述符。所以进程可打开的最大描述符数就限制了我们可以遍历的文件系统树的深度。注意，Single UNIX Specification的XSI扩展中说明的ftw允许调用者指定使用的描述符数，这隐含着可以关闭描述符并且重用它们。

4.12 chroot函数被因特网文件传输程序（Internet File Transfer Program，FTP）用于辅助安全性。系统中没有账号的用户（也称为匿名FTP）放在一个单独的目录下，利用chroot将此目录当作新的根目录就可以阻止用户访问此目录以外的文件。

chroot也用于在另一台机器上构造文件系统层次结构的一个副本，然后修改此副本，

但不更改原来的文件系统。这可用于测试新软件包的安装。

chroot只能由超级用户执行，一旦更改了一个进程的根目录，该进程及其后代进程就再也不能恢复至原先的根目录。

4.13 首先，调用stat函数取得文件的三个时间值，然后调用utime设置期望的值。在调用utime时我们不希望改变的值应当是stat中相应的值。

4.14 finger(1)命令对邮箱调用stat函数，最近一次修改的时间是上一次接收邮件的时间，最近存取时间是上一次读邮件的时间。

4.15 cpio和tar存储的只是归档的修改时间（st_mtime）。因为文件归档时一定会读它，所以该文件的存取时间对应于创建归档的时间，为此没有存放其存取时间。cpio的-a选项可以在读输入文件后重新设置该文件的存取时间，于是创建归档不改变文件的存取时间。（但是，重新设置文件的存取时间确实改变了状态更改时间。）状态更改时间没有存放在文档上，因为在抽取文件时即使它曾被归档，在抽取时也不能设置其值。（utime函数可以更改的仅是存取时间和修改时间。）

对tar来说，在抽取文件时，其默认方式是复原归档时的修改时间值，但是m选项则将修改时间设置为抽取文件时的时间，而不是复原归档时的修改时间值。对于tar，无论何种情况，在抽取后，文件的存取时间均是抽取文件时的时间。 858

另一方面，cpio将存取时间和修改时间设置为抽取文件时的时间。默认情况下，它并不试图将修改时间设置为归档时的值。cpio的-m选项将文件的修改时间和存取时间都设置为归档时的值。

4.16 内核对目录树的深度没有内在的限制，但是如果路径名的长度超出了PATH_MAX，则有许多命令会失败。程序清单C-2中的程序创建了一个深度为100的目录树，每一级目录名有45个字符。在所有平台上我们都能构建这样的结构，但是却不能在所有平台上用getcwd都得到第100级目录的绝对路径名。在Linux 2.4.22和Solaris 9中，当到达长路径的目录结尾时，getcwd就不再能继续正常工作。在FreeBSD 5.2.1和Mac OS X 10.3中，getcwd可以获得路径名，但是需要多次调用realloc得到一个足够大的缓冲区。在FreeBSD 5.2.1上运行该程序后得到

```
$ ./a.out
getcwd failed, size = 1025: Result too large
getcwd failed, size = 1125: Result too large
...                              33 行
getcwd failed, size = 4525: Result too large
length = 4610
                                 显示4610字节的路径名
```

但是由于文件名太长，不能用tar或cpio对该目录建立档案文件，它们都“抱怨”文件名太长了。

程序清单C-2 创建深目录树

```
#include "apue.h"
#include <fcntl.h>

#define DEPTH   100         /* directory depth */
#define MYHOME  "/home/sar"
#define NAME    "alonglonglonglonglonglonglonglonglonglongname"
#define MAXSZ   8192
```

```
int
main(void)
{
    int     i, size;
    char    *path;

    if (chdir(MYHOME) < 0)
        err_sys("chdir error");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("mkdir failed, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("chdir failed, i = %d", i);
    }
    if (creat("afile", FILE_MODE) < 0)
        err_sys("creat error");

    /*
     * The deep directory is created, with a file at the leaf.
     * Now let's try to obtain its pathname.
     */
    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL) {
            break;
        } else {
            err_ret("getcwd failed, size = %d", size);
            size += 100;
            if (size > MAXSZ)
                err_quit("giving up");
            if ((path = realloc(path, size)) == NULL)
                err_sys("realloc error");
        }
    }
    printf("length = %d\n%s\n", strlen(path), path);

    exit(0);
}
```

4.17 /dev目录关闭了一般用户的写访问权限，以避免用户删除目录中的文件，这就意味着unlink失败。

第5章

5.2 fgets函数读入数据，直到行结束或缓冲区满（当然会留出一个字节存放终止字符）。同样，fputs只负责将缓冲区中的内容输出，而并不考虑缓冲区中是否包含换行符。所以，如果将MAXLINE设得太小，这两个函数仍然会正常工作，只不过被执行的次数要比MAXLINE值较大的时候多。

如果这些函数删除或添加换行符（如gets和puts），则必须保证对于最长的行，缓冲区也足够大。

859～860

5.3 当printf没有输出任何字符时，如printf("")，则返回0。

5.4 这是一个比较常见的错误。getc以及getchar的返回值都是整型，而不是字符型。由于EOF经常定义为-1，那么如果系统使用的是有符号的字符类型，程序还可以正常工作。

但如果使用的是无符号字符类型，那么返回的EOF被保存到字符c后将不再是-1，所以，程序会进入死循环。本书描述的四种平台都使用带符号字符，所以实例代码都能工作。

5.5 5个字符长的前缀、4个字符长的进程内唯一标识再加5个字符长的系统内唯一标识（进程ID）刚好组成14位的UNIX传统文件名长度限制。

5.6 使用方法为：先调用fflush后调用fsync，fsync的参数由fileno函数获得。如果不调用fflush，所有的数据仍然在内存缓冲区中，此时调用fsync将没有任何效果。

5.7 当程序交互运行时，标准输入和标准输出均为行缓冲方式。每次调用fgets时标准输出设备将自动刷清。

第6章

6.1 6.3节论述了在Linux和Solaris系统中存取阴影口令文件的函数。我们不能使用6.2节所述函数返回的pw_passwd字段值与加密口令相比较。正确的方法是使用阴影口令文件中对应用户的加密口令来进行比较。

在FreeBSD和Mac OS X，口令文件的阴影是自动建立的。getpwnam或getpwuid函数返回的passed结构中，pw_passwd字段包含加密口令（在FreeBSD，仅当调用者的有效用户ID为0时）。

6.2 在Linux和Solaris中，程序清单C-3中的程序输出加密口令。当然，除非有超级用户权限，否则调用getspnam将返回EACCES错误。

程序清单C-3 在SVR4系统中输出加密口令

```
#include "apue.h"
#include <shadow.h>

int
main(void)      /* Linux/Solaris version */
{
    struct spwd *ptr;

    if ((ptr = getspnam("sar")) == NULL)
        err_sys("getspnam error");
    printf("sp_pwdp = %s\n", ptr->sp_pwdp == NULL ||
      ptr->sp_pwdp[0] == 0 ?  "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

861

在FreeBSD中，若以超级用户权限运行这个程序，程序清单C-4中的程序将输出加密口令，否则pw_passed的返回值为星号（*）。在Mac OS X，不管其运行时的用户权限是什么都输出加密口令。

程序清单C-4 在FreeBSD和Mac OS X中输出加密口令

```
#include "apue.h"
#include <pwd.h>

int
main(void)      /* FreeBSD/Mac OS X version */
{
    struct passwd   *ptr;

    if ((ptr = getpwnam("sar")) == NULL)
        err_sys("getpwnam error");
```

```
    printf("pw_passwd = %s\n", ptr->pw_passwd == NULL ||
      ptr->pw_passwd[0] == 0 ?  "(null)" : ptr->pw_passwd);
    exit(0);
}
```

6.5 程序清单C-5中的程序以类似于date的格式输出日期。

程序清单C-5　以类似于date(1)的格式输出日期和时间

```
#include "apue.h"
#include <time.h>

int
main(void)
{
    time_t      caltime;
    struct tm   *tm;
    char        line[MAXLINE];

    if ((caltime = time(NULL)) == -1)
        err_sys("time error");
    if ((tm = localtime(&caltime)) == NULL)
        err_sys("localtime error");
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("strftime error");
    fputs(line, stdout);
    exit(0);
}
```

程序清单C-5中的程序的运行结果如下：

```
$ ./a.out                              作者的默认格式是在美国东部
Sun Feb 06 16:53:57 EST 2005
$ TZ=US/Mountain ./a.out               美国山地时区
Sun Feb 06 14:53:57 MST 2005
$ TZ=Japan ./a.out                     日本
Mon Feb 07 06:53:57 JST 2005
```

862

第7章

7.1 原因在于printf的返回值（输出的字符数）变成了main函数的返回值。当然，并不是所有的系统都会出现该情况。

7.2 当程序处于交互运行方式时，标准输出通常处于行缓冲方式，所以当输出换行符时，上次的结果才被真正输出。如果标准输出被定向到一个文件而处于完全缓冲方式，则当标准I/O清理操作执行时，结果才真正被输出。

7.3 由于agrc和argv的副本不像environ一样保存在全局变量中，所以在大多数UNIX系统中没有办法做到这一点。

7.4 当C程序解引用一个空指针出错时，这种方法终止进程。

7.5 定义如下：

```
typedef void    Exitfunc(void);

int atexit(Exitfunc *func);
```

7.6 calloc将分配的内存空间初始化为0。但是ISO C并不保证0值与浮点0或空指针的值相同。

7.7 只有通过exec函数执行一个程序时，才会分配堆和堆栈（见8.10节）。

7.8 可执行文件（a.out）包含了用于调试core文件的符号表信息，用strip(1)命令可以删除这些信息，对两个a.out文件执行这条命令，它们的大小分别减为381 976和2 912字节。

7.9 没有使用共享库时，可执行文件的大部分都被标准I/O库所占用。

7.10 这段代码不正确。因为在复合语句中定义了自动变量val，当该复合语句结束时，变量val就不存在了，而这段代码通过指针引用了已经不存在的自动变量val。

第8章

8.1 为了仿真子进程终止时关闭标准输出的行为，在调用exit之前加下面一行：

```
fclose(stdout);
```

为了观察其效果，用下面几行代替程序中调用printf的语句：

```
i = printf("pid = %d, glob = %d, var = %d\n",
    getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

863

注意也要定义变量i和buf。

这里假设子进程调用exit时关闭标准I/O流，并不关闭文件描述符STDOUT_FILENO。有些版本的标准I/O库会关闭与标准输出相关联的文件描述符从而引起write标准输出失败，在这种情况，调用dup将标准输出复制到另一个描述符，write则使用新复制的文件描述符。

8.2 可以通过程序清单C-6中的程序来说明这个问题。

程序清单C-6 错误使用vfork的例子

```
#include "apue.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t   pid;

    if ((pid = vfork()) < 0)
        err_sys("vfork error");
    /* child and parent both return */
}

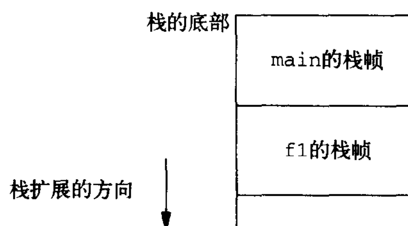
static void
f2(void)
{
    char    buf[1000];      /* automatic variables */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
```

```
        buf[i] = 0;
}
```

当函数f1调用vfork时，父进程的栈指针指向f1函数的栈帧，见图C-2。

vfork使得子进程先执行然后从f1返回，接着子进程调用f2，并且f2的栈帧覆盖了f1的栈帧，在f2中子进程将自动变量buf的值置为0，即将栈中的1000个字节的值都置为0。864从f2返回后子进程调用_exit，这时栈中main栈帧以下的内容已经被f2修改了。然后，父进程从vfork调用后继续，并从f1返回。返回信息虽然常常保存在栈中，但是多半可能已经被子进程修改了。对于这个例子，父进程恢复继续执行的结果要依赖于你所使用的UNIX系统的实现特征。(如返回信息保存在栈帧中的具体位置，修改动态变量时覆盖了哪些信息，等等。)通常的结果是一个core文件，但在你的系统上产生的结果可能不同。



图C-2　调用vfork时的栈帧

8.3 在程序清单8-7中，我们先让父进程输出，但是当父进程输出完毕子进程要输出时，要让父进程终止。是父进程先终止还是子进程先执行输出要依赖于内核对两个进程的调度(另一个竞争条件)。在父进程终止后，shell会开始执行下一个程序，它也许会干扰子进程的输出。为了避免这种情况，要在子进程完成输出后才终止父进程。用下面的语句替换程序中fork后面的代码。

```
else if (pid == 0) {
    WAIT_PARENT();           /* parent goes first */
    charatatime("output from child\n");
    TELL_PARENT(getppid()); /* tell parent we're done */
} else {
    charatatime("output from parent\n");
    TELL_CHILD(pid);         /* tell child we're done */
    WAIT_CHILD();            /* wait for child to finish */
}
```

由于只有终止父进程才能开始下一个程序，而该程序让子进程先运行，所以不会出现上面的情况。

8.4 对argv[2]打印的是相同的值（/home/stevens/bin/testinterp)。原因是865execlp在结束时调用了execve，并且与直接调用execl的*路径名*相同。回忆图8-2。

8.5 不提供返回保存的设置用户ID的函数，我们必须在进程开始时保存有效的用户ID。

8.6 程序清单C-7中的程序创建了一个僵死进程。

程序清单C-7　创建一个僵死进程并用ps查看其状态

```
#include "apue.h"

#ifdef SOLARIS
#define PSCMD    "ps -a -o pid,ppid,s,tty,comm"
#else
```

```
#define PSCMD   "ps -o pid,ppid,state,tty,command"
#endif

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)      /* child */
        exit(0);

    /* parent */
    sleep(4);
    system(PSCMD);

    exit(0);
}
```

执行程序结果如下（ps(1)用Z表示僵死进程）：

```
$ ./a.out
  PID  PPID S TT       COMMAND
 3395  3264 S pts/3    bash
29520  3395 S pts/3    ./a.out
29521 29520 Z pts/3    [a.out] <defunct>
29522 29520 R pts/3    ps -o pid,ppid,state,tty,command
```

第9章

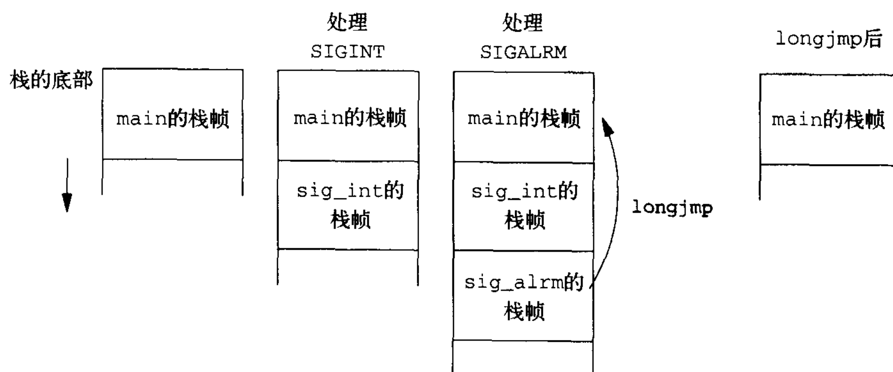
9.1 因为init进程是登录shell的父进程，当登录shell终止时它收到SIGCHLD信号量，所以init进程知道什么时候终端用户注销。

网络登录没有包含init，在utmp和wtmp文件中的登录项和相应的注销项是由一个处理登录并监测注销的进程写的（本例中为telnetd）。 866

第10章

10.1 当程序第一次接收到发送给它的信号量时就终止了。因为一捕捉到信号，pause函数就返回。

10.2 图C-3给出了这些栈帧。



图C-3 longjmp前后的栈帧

在sig_alrm中通过longjmp返回main，有效地避免了继续执行sig_int。

10.4 在第一次调用alarm和调用setjmp之间又有一个竞争条件。如果进程在调用alarm和setjmp之间被内核阻塞了，闹钟时间超过后就调用信号量处理程序，然后调用longjmp。但是由于没有调用过setjmp，所以没有设置env_alrm缓冲区。如果longjmp的跳转缓冲区没有被setjmp初始化，则说明longjmp的操作是未定义的。

10.5 参见Don Libes的“Implementing Software Timers”（*C Users Journal*，vol.8，no.11，Nov.1990）中的例子。

10.7 如果仅仅调用_exit，则进程终止状态不能表示该进程是由于SIGABRT信号量而终止的。

10.8 如果信号是由其他用户的进程发出的，进程必须是设置用户ID为根的或者是接收进程的所有者，否则kill不能执行。所以实际用户ID为信号的接收者提供了更多的信息。

10.10 对于本书作者所用的一个系统，大约每60~90分钟增加一秒，这个误差是因为每次调用sleep都要调度一次将来时间的事件，但是由于CPU调度，有时并没有在事件发生时立即被唤醒。另外一个原因是进程开始运行和再次调用sleep都需要一定量的时间。

867

cron守护进程这样的程序每分钟都要取当前时间，而且它首先设置一个休眠周期，然后在下一分钟开始时唤醒。大多数调用是sleep(60)，偶尔有一个sleep(59)用于在下一分钟同步。但是若在进程中花费了许多时间执行命令或者系统的负载重、调度慢，这时休眠值可能远小于60。

10.11 在Linux 2.4.22和Solaris 9中，从来没有调用过SIGXFSZ的信号量处理程序，但一旦文件的大小达到1024字节时，write就返回24。

在FreeBSD 5.2.1和Mac OS X 10.3中，文件大小已达到1000字节，下一次企图写100字节时调用该信号处理程序，write调用返回−1并且errno设置为EFBIG（“文件太大”）。

10.12 结果与标准I/O库的实现有关：fwrite如何处理一个被中断的写。

第11章

11.1 程序清单C-8给出了一个没有使用自动变量，而采用动态内存分配的程序。

程序清单C-8　线程返回值的正确使用

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf(s);
    printf("  structure at 0x%x\n", (unsigned)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
```

```
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) == NULL)
        err_sys("can't allocate memory");
    fp->a = 1;
    fp->b = 2;
    fp->c = 3;
    fp->d = 4;
    printfoo("thread:\n", fp);
    return((void *)fp);
}

int
main(void)
{
    int err;
    pthread_t tid1;
    struct foo *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printfoo("parent:\n", fp);
    exit(0);
}
```

868

11.2 要修改未决作业的线程ID，必须持有写模式下的读写锁，防止ID在改变过程中有其他线程在搜索该列表。目前定义该接口的方式存在的问题在于：调用job_find找到该作业以及调用job_remove从列表中删除该作业这两个时间之间作业ID可以改动。这个问题可以通过在job结构中嵌入引用计数和互斥量，然后让job_find增加引用计数的方法来解决。这样修改ID的代码就可以避免对列表中非零引用计数的任何作业进行ID改动的情况。

11.3 首先，列表是由读写锁保护的，但条件变量需要互斥量对条件进行保护；其次，每个线程等待满足的条件应该是有某个作业进行处理时需要的条件，所以需要创建每线程数据结构来表示这个条件。或者，可以把互斥量和条件变量嵌入到queue结构，但这意味着所有的工作线程都将等待相同的条件。如果有很多工作线程存在，当唤醒了许多线程但又没有工作可做时，就可能出现*惊群效应*（thundering herd）问题，最后导致CPU资源的浪费，并且增加了锁的争夺。

11.4 这根据具体情况而定。总的来说，两种情况都可能是正确的，但每一种方法都有不足之处。在第一种情况下，调用pthread_cond_broadcast以后，等待线程会被调度以运行。如果程序运行在多处理机上，由于还持有互斥锁（pthread_cond_wait返回持有的互斥锁），一些线程就会运行而且马上阻塞。在第二种情况下，运行线程可以在第3步和第4步之间获取互斥锁，然后使条件失效，最后释放互斥锁；接着，当调用pthread_cond_broadcast时，条件不再为真，线程无需运行。这就是为什么唤醒线程必须重新检查条件，不能仅仅因为pthread_cond_wait返回就假定条件为真。

869

第12章

12.1 就像人们首先会猜到的，这并不是一个多线程问题。这些标准I/O例程事实上是线程安

全的。调用fork时，每个进程获得了标准I/O数据结构的一个副本。程序运行时把标准输出定向到终端时，输出是行缓冲的，所以每次打印一行时，标准I/O库就把该行写到终端上。但是，如果把标准输出重定向到文件的话，则标准输出就是全缓冲的。当缓冲区满或者进程关闭流时，输出才会写到文件。在这个例子中执行fork时，缓冲区中包含了还未写的几个打印行，所以当父进程和子进程最终把缓冲区中的副本冲洗时，最初的复制内容就写入文件。

12.3 理论上来讲，如果在信号处理程序运行时阻塞所有的信号，那么就能使函数成为异步信号安全的。问题是我们并不能知道调用的某个函数可能并没有屏蔽已经被阻塞的信号，这样可能通过另一个信号处理程序使得该函数变成可重入的。

12.4 在FreeBSD 5.2.1上，会得到一连串错误信息，然后过一段时间，程序抛出core文件。用gdb的话可以看到程序在初始化阶段就陷入无限循环。程序初始化过程将调用线程初始化函数，这会调用到malloc。而malloc函数反过来会调用getenv找到环境变量MALLOC_OPTIONS的值。我们对getenv的实现调用了posix线程函数，posix线程函数会试图调用线程初始化函数。最终会出现错误，在调用abort以后陷入类似的无限循环。经过几十万左右的栈帧以后，进程退出并产生core转储。

12.5 如果希望在一个程序中运行另一个程序（即在调用exec之前），还需要fork。

12.6 程序清单C-9给出了使用select实现线程安全的sleep函数，延迟一定数量的时间。它是线程安全的，因为它并不使用任何未经保护的全局或静态数据，并且只调用其他线程安全的函数。

程序清单C-9　sleep的线程安全实现

```
#include <unistd.h>
#include <time.h>
#include <sys/select.h>

unsigned
sleep(unsigned nsec)
{
    int n;
    unsigned slept;
    time_t start, end;
    struct timeval tv;

    tv.tv_sec = nsec;
    tv.tv_usec = 0;
    time(&start);
    n = select(0, NULL, NULL, NULL, &tv);
    if (n == 0)
        return(0);
    time(&end);
    slept = end - start;
    if (slept >= nsec)
        return(0);
    return(nsec - slept);
}
```

12.7 很多时候条件变量的实现都使用互斥锁来保护它的内部结构，由于这是实现细节因而通常是被隐藏起来的，所以在fork处理程序中没有可移植的方法获取或释放锁。既然在调用fork后并不能确定条件变量中的内部锁状态，所以在子进程中使用条件变量是不安全的。

第13章

13.1 如果进程调用chroot，它就不能打开/dev/log。解决的办法是，守护进程在调用chroot之前调用选项为LOG_NDELAY的openlog。它打开特殊设备文件（UNIX域数据报套接字）并生成一个描述符，即使调用了chroot之后，该描述符仍然是有效的。这种情景在ftpd（File Transfer Protocol daemon）这样的守护进程中是会碰到的，为安全性原因，它们调用chroot，但仍需调用syslog以记录出错条件。

13.3 程序清单C-10程序是一种解决方案。其结果依赖于不同的系统实现。请回忆，daemonize关闭所有打开文件描述符，然后向/dev/null再打开前3个。这意味着进程不再有控制终端，所以getlogin不能在utmp文件中看到进程的登录项。于是在Linux 2.4.22和Solaris 9，我们发现守护进程没有登录名。

870 ~ 871

程序清单C-10 调用daemon_init获得注册名

```
#include "apue.h"

int
main(void)
{
    FILE *fp;
    char *p;

    daemonize("getlog");
    p = getlogin();
    fp = fopen("/tmp/getlog.out", "w");
    if (fp != NULL) {
        if (p == NULL)
            fprintf(fp, "no login name\n");
        else
            fprintf(fp, "login name: %s\n", p);
    }
    exit(0);
}
```

但是在FreeBSD 5.2.1和Mac OS X 10.3中，登录名是由进程表维护的，并且在执行fork时复制。也就是说，除非其父进程没有登录名（如系统自引导时调用init），否则进程总能获得其登录名。

第14章

14.1 测试程序示于程序清单C-11。

程序清单C-11 判断记录锁的行为

```
#include "apue.h"
#include <fcntl.h>
#include <errno.h>

void
sigint(int signo)
{
}

int
main(void)
```

```
{
    pid_t pid1, pid2, pid3;
    int fd;

    setbuf(stdout, NULL);
    signal_intr(SIGINT, sigint);

    /*
     * Create a file.
     */
    if ((fd = open("lockfile", O_RDWR|O_CREAT, 0666)) < 0)
        err_sys("can't open/create lockfile");

    /*
     * Read-lock the file.
     */
    if ((pid1 = fork()) < 0) {
        err_sys("fork failed");
    } else if (pid1 == 0) { /* child */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("child 1: can't read-lock file");
        printf("child 1: obtained read lock on file\n");
        pause();
        printf("child 1: exit after pause\n");
        exit(0);
    } else {        /* parent */
        sleep(2);
    }

    /*
     * Parent continues ... read-lock the file again.
     */
    if ((pid2 = fork()) < 0) {
        err_sys("fork failed");
    } else if (pid2 == 0) { /* child */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("child 2: can't read-lock file");
        printf("child 2: obtained read lock on file\n");
        pause();
        printf("child 2: exit after pause\n");
        exit(0);
    } else {        /* parent */
        sleep(2);
    }

    /*
     * Parent continues ... block while trying to write-lock
     * the file.
     */
    if ((pid3 = fork()) < 0) {
        err_sys("fork failed");
    } else if (pid3 == 0) { /* child */
        if (lock_reg(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
            printf("child 3: can't set write lock: %s\n",
              strerror(errno));
        printf("child 3 about to block in write-lock...\n");
        if (lock_reg(fd, F_SETLKW, F_WRLCK, 0, SEEK_SET, 0) < 0)
            err_sys("child 3: can't write-lock file");
        printf("child 3 returned and got write lock????\n");
        pause();
        printf("child 3: exit after pause\n");
        exit(0);
```

872

```
    } else {            /* parent */
        sleep(2);
    }
    /*
     * See if a pending write lock will block the next
     * read-lock attempt.
     */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        printf("parent: can't set read lock: %s\n",
          strerror(errno));
    else
        printf("parent: obtained additional read lock while"
          " write lock is pending\n");
    printf("killing child 1...\n");
    kill(pid1, SIGINT);
    printf("killing child 2...\n");
    kill(pid2, SIGINT);
    printf("killing child 3...\n");
    kill(pid3, SIGINT);
    exit(0);
}
```

873

在本书说明的四种平台上，记录锁的行为是相同的：后增加的读者可使未决的写者不断等待。运行该程序得到

```
child 1: obtained read lock on file
child 2: obtained read lock on file
child 3: can't set write lock: Resource temporarily unavailable
child 3 about to block in write-lock...
parent: obtained additional read lock while write lock is pending
killing child 1...
child 1: exit after pause
killing child 2...
child 2: exit after pause
killing child 3...
child 3: can't write-lock file: Interrupted system call
```

14.2 大多数系统将fd_set定义为只包含一个成员的结构，该成员为一个长整型数组。数组中每一位（bit）对应于一个描述符。4个FD_宏通过开、关或测试指定的位对这个数组进行操作。将之定义为一个包含数组的结构而不仅仅是一个数组的原因是：通过C语言的赋值语句，可以使fd_set类型变量相互赋值。

14.3 大多数系统允许用户在包括头文件<sys/types.h>前定义常量FD_SETSIZE。例如，下面的代码定义fd_set数据类型，使其可以包含2048个描述符：

```
#define FD_SETSIZE  2048
#include <sys/select.h>
```

这在FreeBSD 5.2.1、Mac OS X 10.3和Solaris 9上可正常工作，而Linux 2.4.22对此的处理则不同。

874

14.4 下面的表中列出了功能类似的函数。

FD_ZERO	sigemptyset
FD_SET	sigaddset
FD_CLR	sigdelset
FD_ISSET	sigismember

没有与sigfillset对应的FD_xxx函数。对信号量集来说，指向信号量集的指针总是第一个参数，信号编号是第二个参数；对于描述符来说，描述符编号是第一个参数，指向描述符集合的指针是第二个参数。

14.5 getmsg最多返回5种信息：数据、数据长度、控制信息、控制信息的长度和标志。

14.6 利用select实现的程序见程序清单C-12，利用poll实现的程序见程序清单C-13。

程序清单C-12　用select实现sleep_us函数

```
#include "apue.h"
#include <sys/select.h>

void
sleep_us(unsigned int nusecs)
{
    struct timeval  tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

程序清单C-13　用poll实现sleep_us函数

```
#include <poll.h>

void
sleep_us(unsigned int nusecs)
{
    struct pollfd   dummy;
    int             timeout;

    if ((timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

如BSD usleep(3)手册页中所说明的，usleep(3)使用setitimer设置间隔计时器，并且每次调用它时，执行8个系统调用。它可以正确地和调用进程设置的其他计时器交互，而且即使捕捉到信号也不会被中断。

875

14.7 不行。我们可以使TELL_WAIT创建一个临时文件，其中一个字节用作为父进程的锁，另一个字节用作为子进程的锁。WAIT_CHILD使得父进程等待获取子进程字节上的锁，TELL_PARENT使得子进程释放子进程字节上的锁。但是问题在于，调用fork导致释放所有子进程中的锁，使得子进程开始运行时不具有任何它自己的锁。

14.8 程序清单C-14中示出了一种解决方法。

程序清单C-14　用非阻塞写计算管道的容量

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    int i, n;
    int fd[2];

    if (pipe(fd) < 0)
```

```
        err_sys("pipe error");
    set_fl(fd[1], O_NONBLOCK);

    /*
     * Write 1 byte at a time until pipe is full.
     */
    for (n = 0; ; n++) {
        if ((i = write(fd[1], "a", 1)) != 1) {
            printf("write ret %d, ", i);
            break;
        }
    }
    printf("pipe capacity = %d\n", n);
    exit(0);
}
```

对于我们说明的四种平台上，计算出的值示于下面的表中。

平　台	管道容量（字节）
FreeBSD 5.2.1	16,384
Linux 2.4.22	4,096
Mac OS X 10.3	8,192
Solaris 9	9,216

这些值可能与对应的PIPE_BUF值不同，其原因是，PIPE_BUF被定义为可被原子地写至一个管道的最大数据量。这里，我们计算的是一个管道独立于任何原子性限制可保持的数据量。

14.10 程序清单14-12中的程序是否更新输入文件的最近一次存取时间依赖于操作系统以及文件所属的文件系统的类型。

876

第15章

15.1 如果写管道端总是不关闭，则读者就决不会看到文件的结束符。分页程序就会一直阻塞在读标准输入。

15.2 父进程向管道写完最后一行以后就终止，当父进程终止时管道的读端自动关闭。但是由于子进程（分页程序）要等待输出的页，所以父进程可能比子进程领先一个管道缓冲区。如果正在运行一个可对命令行进行编辑的交互式shell上，如Korn shell,当父进程终止时shell多半会改变终端的模式并提示用户。（由于大部分分页程序在等待处理下一个页面时将终端设置为非正规模式，所以这无疑会影响分页程序。）

15.3 因为执行了shell，所以popen函数返回一个文件指针。但是shell不能执行不存在的命令，因此在标准错误上显示下面信息后终止，其退出状态为127，pclose返回该命令的终止状态，这如同从waitpid返回一样：

```
sh: line 1: ./a.out: No such file or directory
```

15.4 当父进程终止时，用shell看它的终止状态。对于Bourne shell、Bourne-again shell和Korn shell，所用的命令是echo $?。打印的结果是128加信号数。

15.5 首先加入下面的声明，

```
FILE    *fpin, *fpout;
```

然后用fdopen关联管道描述符和标准I/O流，并将流设置为行缓冲的。在从标准输入读

的while循环之前做此工作。

```
    if ((fpin = fdopen(fd2[0], "r")) == NULL)
        err_sys("fdopen error");
    if ((fpout = fdopen(fd1[1], "w")) == NULL)
        err_sys("fdopen error");
    if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
        err_sys("setvbuf error");
    if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
        err_sys("setvbuf error");
```

while中的read和write用下面的语句代替：

```
    if (fputs(line, fpout) == EOF)
        err_sys("fputs error to pipe");
    if (fgets(line, MAXLINE, fpin) == NULL) {
        err_msg("child closed pipe");
        break;
    }
```

877

15.6 system函数调用了wait，终止的第一个子进程是由popen产生的。因为该子进程不是system创建的，所以它将再次调用wait并一直阻塞到sleep完成。然后system返回。当pclose调用wait时，由于没有子进程可等待所以返回出错，导致pclose也返回出错。

15.7 select表明描述符是可读的。调用read读完所有的数据后返回0就表明到达了文件结尾。但是对于poll（假设管道是基于STREAMS的）来说，若返回POLLHUP，那么也许仍有数据可以读。但是一旦读完了所有的数据read就返回0，即表明到达了文件结尾。在读完了所有的数据后，即使需安再调用一次read以接收文件结尾通知也不返回POLLIN。

对于已被读者关闭的引用管道的输出描述符来说，select表明该描述符是可写的。但当调用write时产生SIGPIPE信号量。如果忽略该信号量或从信号量处理程序中返回时，write就返回EPIPE错误。而对于poll，如果管道是基于STREAMS的，poll就对该描述符返回POLLHUP。

15.8 子进程向标准出错写的内容同样也在父进程的标准出错中出现。只要在*cmdstring*中包含重定向2>&1，就可以将标准出错发送给父进程。

15.9 popen函数fork一个子进程，子进程通过exec执行Bourne shell。然后shell再调用fork，最后由shell的子进程执行命令串。当*cmdstring*终止时shell恰好在等待该事件，然后shell退出，而这一事件又是pclose中的waitpid所等待的。

15.10 解决的办法是打开FIFO两次，一次读一次写。我们绝不会使用为写而打开的描述符，但是使该描述符打开就可在客户数从1变为0时，阻止产生文件终止。打开FIFO两次需要注意下列操作方式（如非阻塞open所要求的）：第一次以非阻塞、只读方式open，第二次以阻塞、只写方式open。（如果先用非阻塞、只写方式open将返回错误。）然后关闭读描述符的非阻塞属性。代码参见程序清单C-15。

程序清单C-15　以非阻塞方式打开FIFO进行读写操作

```
#include "apue.h"
#include <fcntl.h>

#define FIFO    "temp.fifo"

int
main(void)
{
    int     fdread, fdwrite;
```

```
    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");
    if ((fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ((fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");
    clr_fl(fdread, O_NONBLOCK);
    exit(0);
}
```

878

15.11 随意读取现行队列中的消息会干扰客户－服务器协议，导致丢失客户请求或者服务器的响应。只要知道队列的标识符或者该队列允许所有的用户读，进程就可以读队列。

15.13 由于服务器和各客户进程可能将段连接到不同的地址，所以在共享存储段中决不会存放实际物理地址。相反，当在共享存储段中建立链表时，指针的值设置为共享存储段内另一对象的偏移量。偏移量为所指对象的实际地址减去共享存储段的起始地址。

15.14 表C-1显示了相关的事件。

表C-1 程序清单15-12中父子进程间的交替过程

父进程i设置为	子进程i设置为	共享值设置为	update返回	说　明
		0		由mmap初始化
	1			子进程先运行，然后阻塞
0				父进程运行
		1		
			0	然后父进程阻塞
		2		子进程恢复
			1	
	3			然后子进程阻塞
2				父进程恢复
		3		
			2	然后父进程阻塞
		4		
			3	
	5			然后子进程阻塞
4				父进程恢复

第16章

16.1 程序清单C-16显示了一个打印系统字节序的程序。

程序清单C-16 判断系统字节序

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int
main(void)
{
    uint32_t i;
    unsigned char *cp;
```

```
    i = 0x04030201;
    cp = (unsigned char *)&i;
    if (*cp == 1)
        printf("little-endian\n");
    else if (*cp == 4)
        printf("big-endian\n");
    else
        printf("who knows?\n");
    exit(0);
}
```

16.3 对于监听的每个端点，需要绑定到一个合适的地址，并对应每个描述符在fd_set中写一条记录。然后使用select等待从多个端点来的连接请求。回忆16.4节，当一个连接请求达到时，一个端点将会变为可读。当一个连接请求实际到达时，接受该请求，并像以前那样进行处理。

16.5 在main函数中，通过调用signal函数（程序清单10-12）来捕获SIGCHLD，该函数使用sigaction来安装可重起的系统调用句柄。下一步，从serve函数中删除waitpid调用。当fork完子进程来处理请求后，父进程关闭新的文件描述符并继续监听新的连接请求。最后，需要一个处理SIGCHLD的信号处理程序如下：

```
void
sigchld(int signo)
{
    while (waitpid((pid_t)-1, NULL, WNOHANG) > 0)
        ;
}
```

16.6 为了启用异步套接字I/O，需要使用F_SETOWN fcntl命令来建立套接字的所有权（ownership），然后使用FIOASYNC ioctl命令启用异步信号即可。为了禁止异步套接字I/O，只要简单地禁止异步信号。之所以混合使用fcntl和ioctl命令的理由是想找到一个可移植的方法来处理异步套接字。代码见程序清单C-17中。

879～880

程序清单C-17　启用与禁止异步套接字I/O

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#if defined(BSD) || defined(MACOS) || defined(SOLARIS)
#include <sys/filio.h>
#endif

int
setasync(int sockfd)
{
    int n;

    if (fcntl(sockfd, F_SETOWN, getpid()) < 0)
        return(-1);
    n = 1;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}
```

```
int
clrasync(int sockfd)
{
    int n;

    n = 0;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}
```

第17章

17.3 声明（declaration）指定了标识符集合的属性（如数据类型），如果声明也造成分配了存储单元，那么这就是定义（definition）。

在头文件opend.h中，用extern存储类声明了三个全局变量，这时并没有为它们分配存储单元。在文件main.c中定义了三个全局变量，有时会在定义时就初始化全局变量，但通常使用C的默认值。

17.5 select和poll作为函数值返回就绪的描述符个数。当将这些就绪描述符都处理完后，操作client数组的循环就可以结束。 881

第18章

18.1 注意由于终端是非规范模式，所以要用换行符而不是回车符终止reset命令。

18.2 它为128个字符建立一张表，根据用户的要求设置奇偶校验位。然后使用8位I/O处理奇偶位的产生。

18.3 如果你使用的是窗口终端，那么你无需登录两次。在两个分开的窗口之间，你可以做这样的实验。在Solaris，运行stty -a，并且将标准输入重定向到运行vi的终端，结果显示vi设置MIN为1、TIME为1。read等待至少敲入一个字符，但是该字符输入后，只对后继的字符等待十分之一秒即返回。

第19章

19.1 telnetd和rlogind两个服务器进程均以超级用户权限运行，所以它们都可以成功地调用chown和chmod。

19.3 执行pty -n stty -a以避免伪终端从设备的termios结构和winsize结构初始化。

19.5 很不幸，fcntl的F_SETFL命令不允许改变读写状态。

19.6 有三个进程组：(1) 登录shell，(2) pty父进程和子进程，(3) cat进程。前两个进程组组成了一个会话，其中，登录shell为会话首进程。第二个会话仅包含cat进程。第一个进程组（登录shell）是后台进程组，其他两个进程组是前台进程组。

19.7 当从其行规程模块接收到文件终止符时，首先是cat终止。它造成PTY从设备终止，这又造成PTY主设备终止。接着，对于正从PTY主设备读取的pty父进程产生一个文件终止符，该父进程将SIGTERM信号发送给子进程，于是子进程终止（子进程不捕捉该信号）。最后，父进程调用main函数结尾的exit(0)。

程序清单8-17中的程序的相关输出为：

```
cat       e =    270, chars =    274, stat =   0:
pty       e =    262, chars =     40, stat =  15: F     X
pty       e =    288, chars =    188, stat =   0:
```

882

19.8 这可通过使用shell的echo和date(1)命令实现：

```
#!/bin/sh
( echo "Script started on " `date`;
  pty "${SHELL:-/bin/sh}";
  echo "Script done on " `date` ) | tee typescript
```

19.9 PTY从设备之上的行规程能够回送，所以pty从其标准输入所读取的以及写向PTY主设备的按默认都回送。尽管程序（ttyname）从不读取数据，但是该回送也可通过从设备之上的行规程模块实现。

第20章

20.1 _db_dodelete中保守的加锁操作是为了避免和db_nextrec发生竞争条件。如果没有使用写锁保护_db_writedat调用，则有可能在_db_nextrec读某个记录时，该记录已被删除：db_nextrec首先读入一个索引记录，判定该记录非空，接着读数据记录，但是在它调用_db_readidx和_db_readdat之间，该记录却可能已被_db_dodelete删除了。

20.2 假定db_nextrec调用_db_readidx，它将记录的关键字读入索引缓冲区。然后，该进程被内核调度进程暂停，另一个进程运行，它刚好调用db_delete删除了这一条记录，使得索引文件和数据记录文件中对应部分都被清空。当第一个进程恢复执行并调用_db_readdat（在db_nextrec函数体中）时，返回的是空数据记录。db_nextrec中的读锁使得读入索引记录的过程和读入数据记录的过程是一个原子操作（对于其他操作同一数据库的合作进程而言）。

20.3 强制锁对其他的读者和写者产生了影响。在_db_writeidx和_db_writedat设置的锁被释放之前，其他的读操作和写操作都将被阻塞。

20.5 在写索引记录之前写数据记录，依靠这一方法来防止如下情形：若该进程在两次写之间被杀死，就会产生不正常的记录。如果进程先写索引记录，而在写数据记录之前被杀死，那么就会得到一个有效的索引记录，但它却指向一个无效的数据记录。

第21章

21.5 这里有一些提示。有两个地方可以检查队列中的作业：打印守护进程的队列和网络打印机的内部队列。注意，不要让一个用户能够取消另外一个用户的打印作业，当然，超级用户可以取消任何作业。

883

参考书目

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. 1986. "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the 1986 Summer USENIX Conference*, pp. 93–113, Atlanta, GA.

介绍 Mach操作系统的一篇文章。

Adobe Systems Inc. 1999. *PostScript Language Reference Manual, Third Edition*. Addison-Wesley, Reading, MA.

PostScript的语言参考手册。

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1988. *The AWK Programming Language*. Addison-Wesley, Reading, MA.

该书对awk程序设计语言进行了完整的说明。该书所描述的awk有时被称为nawk（new awk）。

Andrade, J. M., Carges, M. T., and Kovach, K. R. 1989. "Building a Transaction Processing System on UNIX Systems," *Proceedings of the 1989 USENIX Transaction Processing Workshop*, vol. May, pp. 13–22, Pittsburgh, PA.

描述AT&T Tuxedo事务处理系统。

Arnold, J. Q. 1986. "Shared Libraries on UNIX System V," *Proceedings of the 1986 Summer USENIX Conference*, pp. 395–404, Atlanta, GA.

描述SVR3中共享库的实现。

AT&T. 1989. *System V Interface Definition, Third Edition*. Addison-Wesley, Reading, MA.

该书为四卷本，说明系统V的源代码界面和运行时的行为。其第3版对应于SVR4。1991年出版了第5卷，它包含了第1~4卷中更新的命令和函数部分。现已绝版。

AT&T. 1990a. *UNIX Research System Programmer's Manual, Tenth Edition, Volume I*. Saunders College Publishing, Fort Worth, TX.

这是Research UNIX系统第10版（V10）的《UNIX程序员手册》。它包含了传统的UNIX系统手册页（第1~9部分）。

AT&T. 1990b. *UNIX Research System Papers, Tenth Edition, Volume II*. Saunders College Publishing, Fort Worth, TX.

Research UNIX系统第10版（V10）第2卷，它包含了说明该系统各个方面的40篇文章。

AT&T. 1990c. *UNIX System V Release 4 BSD/XENIX Compatibility Guide*. Prentice-Hall, Englewood Cliffs, NJ.

包含描述兼容库的手册页。

AT&T. 1990d. *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice-Hall, Englewood Cliffs, NJ.

说明SVR4的STREAMS（流）系统。

AT&T. 1990e. *UNIX System V Release 4 Programmer's Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ.

该书是针对Intel 80386处理器的SVR4实现的程序员参考手册。它包含第1部分（命令）、第2部分（系统调用）、第3部分（子例程）、第4部分（文件格式）和第5部分（其他设施）。

AT&T. 1991. *UNIX System V Release 4 System Administrator's Reference Manual.* Prentice-Hall, Englewood Cliffs, NJ.

该书是针对Intel 80386处理器的SVR4实现的管理员参考手册。它包含第1部分（命令）、第4部分（文件格式）、第5部分（其他设施）和第7部分（特殊文件）。

Bach, M. J. 1986. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, NJ.

该书详细描述UNIX操作系统的设计和实现。虽然该书并未提供UNIX系统源代码（因为这当时是AT&T的财产），但提供并讨论了UNIX内核使用的很多算法及数据结构。该书描述的是SVR2。

Bolsky, M. I., and Korn, D. G. 1995. *The New KornShell Command and Programming Language, Second Edition.* Prentice-Hall, Englewood Cliffs, NJ.

说明如何使用作为命令解释器和编程语言的Korn shell。

Chen, D., Barkley, R. E., and Lee, T. P. 1990. "Insuring Improved VM Performance: Some No-Fault Policies," *Proceedings of the 1990 Winter USENIX Conference*, pp. 11–22, Washington, D.C.

该书描述对SVR4虚拟存储器实现的更改，其目的是改善其性能，特别是fork和exec的性能。

Comer, D. E. 1979. "The Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137 (June).

关于B树的一篇很好的综述性文章。

Date, C. J. 2004. *An Introduction to Database Systems, Eighth Edition.* Addison-Wesley, Boston, MA.

对数据库系统的全面概述。

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. 1979. "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Transactions on Databases*, vol. 4, no. 3, pp. 315–344 (September).

886

描述可扩展散列技术的一篇文章。

Fowler, G. S., Korn, D. G., and Vo, K. P. 1989. "An Efficient File Hierarchy Walker," *Proceeding of the 1989 Summer USENIX Conference*, pp. 173–188, Baltimore, MD.

描述一个替代的库函数，其作用是遍历文件系统层次结构。

Gallmeister, B. O. 1995. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Sebastopol, CA.

描述POSIX标准中的实时接口。

Garfinkel, S., Spafford, G., and Schwartz, A. 2003. *Practical UNIX & Internet Security, Third Edition.* O'Reilly & Associates, Sebastopol, CA.

该书详细描述UNIX系统的安全性。

Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. 1987. "Shared Libraries in SunOS," *Proceedings of the 1987 Summer USENIX Conference*, pp. 131–145, Phoenix, AZ.

描述SunOS共享库的实现。

Gingell, R. A., Moran, J. P., and Shannon, W. A. 1987. "Virtual Memory Architecture in SunOS," *Proceedings of the 1987 Summer USENIX Conference*, pp. 81–94, Phoenix, AZ.

描述mmap函数的初始实现和虚拟存储器设计中的有关问题。

Goodheart, B. 1991. *UNIX Curses Explained.* Prentice-Hall, Englewood Cliffs, NJ.

该书详细说明terminfo和curses函数库。现已绝版。

Hume, A. G. 1988. "A Tale of Two Greps," *Software Practice and Experience*, vol. 18, no. 11, pp. 1063–1072.

讨论grep性能改进的一篇有价值的论文。

IEEE. 1990. *Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]*. IEEE (Dec.).

这是第一个POSIX标准，它定义了基于UNIX操作系统的C语言系统接口标准。这常被称为POSIX.1.

ISO. 1999. *International Standard ISO/IEC 9899—Programming Language C*. ISO/IEC.

C语言及标准函数库的官方标准。

此标准的PDF版可在线在http://www.ansi.org 或 http://www.iso.org定购。

Kernighan, B. W., and Pike, R. 1984. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, NJ.

该书是关于UNIX程序设计附加细节的参考书，书中包含了许多UNIX命令和实用程序，如grep、sed、awk以及Bourne shell。

Kernighan, B. W., and Ritchie, D. M. 1988. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ.

该书说明C程序设计语言的ANSI标准。附录B中包含了对ANSI标准定义的函数库的描述。

Kleiman, S. R. 1986. "Vnodes: An Architecture for Multiple File System Types in Sun Unix," *Proceedings of the 1986 Summer USENIX Conference*, pp. 238–247, Atlanta, GA.

描述了原始的v节点实现。

887

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*. Addison-Wesley, Boston, MA.

描述排序和搜索算法。

Korn, D. G., and Vo, K. P. 1991. "SFIO: Safe/Fast String/File IO," *Proceedings of the 1991 Summer USENIX Conference*, pp. 235–255, Nashville, TN.

描述了标准I/O函数库的一种替换软件。该库可在http://www.research.att.com/sw/tools/sfio获得。

Krieger, O., Stumm, M., and Unrau, R. 1992. "Exploiting the Advantages of Mapped Files for Stream I/O," *Proceedings of the 1992 Winter USENIX Conference*, pp. 27–42, San Francisco, CA.

一种标准I/O函数库的替换软件，它基于映射文件。

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA.

该书对4.3BSD UNIX系统进行完整的说明，书中描述的是4.3BSD的Tahoe版。现已绝版。

Lennert, D. 1987. "How to Write a UNIX Daemon," *;login:*, vol. 12, no. 4, pp. 17–23 (July/August).

描述如何编写UNIX系统中的守护进程。

Libes, D. 1990. "expect: Curing Those Uncontrollable Fits of Interaction," *Proceedings of the 1990 Summer USENIX Conference*, pp. 183–192, Anaheim, CA.

对expect程序及其实现的描述。

Libes, D. 1991. "expect: Scripts for Controlling Interactive Processes," *Computing Systems*, vol. 4, no. 2, pp. 99–125 (Spring).

本文提供了很多expect脚本。

Libes, D. 1994. *Exploring Expect*. O'Reilly & Associates, Sebastopol, CA.

使用expect程序的一本全书。

Lions, J. 1977. *A Commentary on the UNIX Operating System*. AT&T Bell Laboratories, Murray Hill, NJ.

描述UNIX System第6版的源代码。仅供AT&T的员工、承包商及内部使用，但在AT&T之外也有大量副本流传。

Lions, J. 1996. *Lions' Commentary on UNIX 6th Edition*. Peer-to-Peer Communications, San Jose, CA.

描述UNIX System第6版的源代码，是1977经典著作的公开可用版。

Litwin, W. 1980. "Linear Hashing: A New Tool for File and Table Addressing," *Proceedings of the 6th International Conference on Very Large Databases*, pp. 212–223, Montreal, Canada.

描述线性散列技术一篇文章。

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, MA.

一本完整描述4.4BSD操作系统的著作。

McKusick, M. K., and Neville-Neil, G. V. 2005. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, Boston, MA.

888

一本完整描述FreeBSD 5.2版操作系统的著作。

Mauro, J., and McDougall, R. 2001. *Solaris Internals*. Prentice-Hall, Upper Saddle River, NJ.

该书对Solaris操作系统的内部技术进行描述，涉及Solaris 2.5.1、2.6和2.7版（也称为Solaris 7）。

Morris, R., and Thompson, K. 1979. "UNIX Password Security," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597 (Nov.).

描述UNIX口令方案设计的历史演变。

Nemeth, E., Snyder, G., Seebass, S., and Hein, T. R. 2001. *UNIX System Administration Handbook, Third Edition*. Prentice-Hall, Upper Saddle River, NJ.

描述了如何管理UNIX系统的一本书。

Olander, D. J., McGrath, G. J., and Israel, R. K. 1986. "A Framework for Networking in System V," *Proceedings of the 1986 Summer USENIX Conference*, pp. 38–45, Atlanta, GA.

本文描述系统V的服务界面、STREAMS和TLI的原始实现。

The Open Group. 2004. *The Single UNIX Specification, Version 3*. The Open Group, Berkshire, UK.

POSIX和X/Open标准组合成一个参考手册。
其HTML版可在`http://www.opengroup.org`上免费取阅，而包含完整标准的CD-ROM则需购买。

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. 1995. "Plan 9 from Bell Labs," *Plan 9 Programmer's Manual Volume 2*. AT&T, Reading, MA.

该书对Plan 9操作系统进行了描述，Plan 9是由研发UNIX系统的同一部门开发的。

Plauger, P. J. 1992. *The Standard C Library*. Prentice-Hall, Englewood Cliffs, NJ.

该书是一本ANSI C函数库的全书，包含了该库的完整C语言实现。

Presotto, D. L., and Ritchie, D. M. 1990. "Interprocess Communication in the Ninth Edition UNIX System," *Software Practice and Experience*, vol. 20, no. S1, pp. S1/3–S1/17 (June).

本文描述Research UNIX系统第9版提供的IPC设施，它是由AT&T贝尔实验室开发的。这种IPC的基础是流输入输出系统，它也包括全双工管道，进程之间传送文件描述符的能力，还包括对服务器的唯一客户连接。本文的副本也刊载在AT&T[1990b]。

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, MA.

该书对UNIX系统V第4版的基于STREAMS的网络编程环境进行描述。

Raymond, E. S., ed. 1996. *The New Hacker's Dictionary, Third Edition*. MIT Press, Cambridge, MA.

定义了大量计算机骇客的术语。

Ritchie, D. M. 1984. "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910 (Oct.).

关于流的第一篇文章。

Salus, P. H. 1994. *A Quarter Century of UNIX*. Addison-Wesley, Reading, MA.

从1969年至1994年的UNIX系统的历史。

Seltzer, M., and Olson, M. 1992. "LIBTP: Portable Modular Transactions for UNIX," *Proceedings of the 1992 Winter USENIX Conference*, pp. 9–25, San Francisco, CA.

描述对db(3)库的修改，它来自于实现了事务的4.3 BSD。

889

Seltzer, M., and Yigit, O. 1991. "A New Hashing Package for UNIX," *Proceedings of the 1991 Winter USENIX Conference*, pp. 173–184, Dallas, TX.

dtm(3)库及它的各种实现的描述，以及一种新的散列处理软件包。

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, NJ.

详细说明UNIX系统下的网络编程。该书的后续版本的内容与其第1版有很大变动。

Stevens, W. R., Fenner, B., and Rudoff, A. M. 2004. *UNIX Network Programming, Volume 1, Third Edition*. Addison-Wesley, Boston, MA.

详细说明UNIX系统下的网络编程。其第2版进行了重新设计并分成两卷，第3版则做了更新。

Stonebraker, M. R. 1981. "Operating System Support for Database Management," *Communications of the ACM*, vol. 24, no. 7, pp. 412–418 (July).

描述操作系统的服务以及它们如何对数据库操作产生影响。

Strang, J. 1986. *Programming with curses*. O'Reilly & Associates, Sebastopol, CA.

一本有关curses的Berkeley版本的书。

Strang, J., Mui, L., and O'Reilly, T. 1988. *termcap & terminfo, Third Edition*. O'Reilly & Associates, Sebastopol, CA.

一本有关termcap和terminfo的书。

Sun Microsystems. 2002. *STREAMS Programming Guide*. Sun Microsystems, Santa Clara, CA.

描述Solaris平台上的STREAMS编程。

Thompson, K. 1978. "UNIX Implementation," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1931–1946 (July–Aug.).

描述UNIX V7的某些实现细节。

Vo, Kiem-Phong. 1996. "Vmalloc: A General and Efficient Memory Allocator," *Software Practice and Experience*, vol. 26, no. 3, pp. 357–374.

描述一种灵活的存储分配器。

Weinberger, P. J. 1982. "Making UNIX Operating Systems Safe for Databases," *The Bell System Technical Journal*, vol. 61, no. 9, pp. 2407–2422 (Nov.).

描述在早期UNIX系统中实现数据库的某些问题。

Weinstock, C. B., and Wulf, W. A. 1988. "Quick Fit: An Efficient Algorithm for Heap Storage Allocation," *SIGPLAN Notices*, vol. 23, no. 10, pp. 141–148.

描述了适用于各种应用程序的存储分配算法。

Williams, T. 1989. "Session Management in System V Release 4," *Proceedings of the 1989 Winter USENIX Conference*, pp. 365–375, San Diego, CA.

描述POSIX.1所基于的SVR4中实现的会话体系结构，包括进程组、作业控制和控制终端。该书还描述了现存方法的安全性方面。

X/Open. 1989. *X/Open Portability Guide*. Prentice-Hall, Englewood Cliffs, NJ.

该书为七卷本，包括命令和实用程序（卷1）、系统接口和头文件（卷2）、补充定义（卷3）、程序设计语言（卷4）、数据管理（卷5）、窗口管理（卷6）以及网络服务（卷7）。虽然该书现已绝版，但现在它已被Single UNIX Specification [Open Group 2004]取代。

890

索 引

标有"definition of"的函数子项指向函数原型出现的地方，当该函数可用时，指向函数的源代码。文中定义的用于后续例子中的函数也包含在索引中，例如程序清单3-5中的set_fl函数。较大例子（第17、19、20和21章）中的某些部分是外部函数的定义，为了便于理解这些大例子，这些外部函数的定义也包含在本索引中。另外，本索引还包括许多例子中出现的重要函数和常量，如select和poll。不过几乎每个例子中都会出现的一般函数（如exit）出现在例子中时并没有为它们建立索引。

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

B

C

D

E

F

G

M

P

Q

R

S

T

U